DOCUMENT RESUME

ED 061 975 52 LI 003 645

AUTHOR	Shoffner, Ralph M., Ed.; Cunningham, Jay L., Ed.
TITLE	The Organization and Search of Bibliographic Records: Component Studies. Final Report.
INSTITUTION	California Univ., Berkeley. Inst. of Library Research.
SPONS AGENCY	Office of Education (DHEW), Washington, D.C. Bureau of Research.
BUREAU NO	BR-7-1083
PUB DATE	Sep 71
GRANT	OEG-1-7-071083-5068
NOTE	317p.;(32 References)
EDRS PRICE	MF-$0.65 HC-$13.16
DESCRIPTORS	*Bibliographic Citations; Indexing; *Information Retrieval; *Information Storage; *Information Systems; *On Line Systems; Program Design; Search Strategies
IDENTIFIERS	Berkeley; *University of California

ABSTRACT

Contained in this report are the results of the second phase (July, 1968 - June, 1970) of the File Organization Project, directed toward the development of a facility in which the many issues relating to the organization and search of bibliographic records in on-line computer environments could be studied. The papers in this volume deal specifically with issues and problems of organizing and accessing large bibliographic files, and are entitled: (1) An analysis of the search problem in files of object descriptions; (2) The assignment of index terms; (3) Design of file structures for on-line bibliographic control systems; (4) The analysis of bibliographic structures, using indexed sequential organization; (5) First-stage model of the economic effects of incorporating a data compression system into an on-line direct-access storage and retrieval system; (6) Implementation of bibliographic record compression; (7) Specification for format translation of the Santa Cruz file; (8) CRUNCH: the Santa Cruz file translation system; (9) Prospects for automatic field recognition; and (10) Translation from ILR processing format to MARC II communications format. [Related documents are LI 003610, LI 003611 and LI 003646 through LI 003648.] (Author/SJ)

U.S. DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
OFFICE OF EDUCATION
THIS DOCUMENT HAS BEEH REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL OFFICE OF EDUCATION POSITION OR POLICY

BR-7-1083
PA-52

FINAL REPORT
Project No. 7-1083
Grant No. OEG-1-7-071083-5068

ED 061975

THE ORGANIZATION AND SEARCH OF BIBLIOGRAPHIC RECORDS: COMPONENT STUDIES

Edited by

Ralph M. Shoffner
Jay L. Cunningham

Institute of Library Research
University of California
Berkeley, California 94720

September 1971

The research reported herein was performed pursuant to a grant with the Office of Education, U.S. Department of Health, Education, and Welfare. Contractors undertaking such projects under Government sponsorship are encouraged to express freely their professional judgment in the conduct of the project. Points of view or opinions stated do not, therefore, necessarily represent official Office of Education position or policy.

LI 003 645

U.S. DEPARTMENT OF
HEALTH, EDUCATION, AND WELFARE

Office of Education
Bureau of Research

TABLE OF CONTENTS

Page

1. AN ANALYSIS OF THE SEARCH PROBLEM IN FILES OF OBJECT DESCRIPTIONS by Irene Travis 1

1.1 The Purpose and Scope of This Chapter......................1
1.2 A General View of Object Description Files..................1
1.2.1 Data Structure and Data Processing...................1
1.2.2 Object Descriptions - Entities, Attributes, and Values..3
1.2.2.1 The Domain of Discourse.....................3
1.2.2.2 The Attribute List..........................5
1.2.2.3 Attributes and Record Formats...............6
1.2.2.4 Value Sets and Record Fields................7
1.3 Searching Directories..8
1.3.1 The Query for a Directory Search.....................8
1.3.2 Retrieval Failures...................................9
1.3.3 Compensating for Formal Errors, Content Errors, and Variants.....................................10
1.4 Similarity Between Value Sets...............................11
1.4.1 Definition...11
1.4.2 The Three Types of Similarity between Value Sets.....13
1.4.2.1 Distributional Similarity...................13
1.4.2.2 Functional Similarity.......................13
1.4.2.3 Elemental Similarity........................16
1.5 Research...20
REFERENCES...21

2. THE ASSIGNMENT OF INDEX TERMS by Marcia Bates. 23

2.1 Summary of Study..23
2.2 Description of Study..23
2.2.1 The Hypotheses.......................................23
2.2.2 Background...25
2.2.3 The Data Base..26
2.2.4 Tests on the Hypotheses..............................27
2.2.4.1 Basic Data on the File......................27
2.2.4.2 The Minor Hypotheses........................33
2.2.4.3 The Major Hypothesis........................37
2.2.5 What Next?...42
2.3 Specific Tests and Data......................................43
2.3.1 Precise Description of Tests Made....................43
2.3.1.1 Basic Data on the File......................43
2.3.1.2 First Minor Hypothesis: Rate of Addition of New Terms......................46
2.3.1.3 Second Minor Hypothesis: Relative Ages of Subject Fields..........................46
2.3.1.4 The Major Hypothesis........................47
2.3.2 Raw Data from Tests..................................49
2.3.2.1 Basic Data on the File......................49
2.3.2.2 First Minor Hypothesis: Rate of Addition of New Terms......................49

TABLE OF CONTENTS (Cont.)

Page

2.3.2.3 Second Minor Hypothesis: Relative Ages of Subject Fields....49
2.3.2.4 Major Hypothesis....50

3. DESIGN OF FILE STRUCTURES FOR ON-LINE BIBLIOGRAPHIC CONTROL SYSTEMS by Jorge Rodriguez . . . 63

3.1 Introduction....63
3.1.1 Design Requirements for Information Retrieval Systems....63
3.1.2 Objective of File Structures Analysis Task....64
3.2 General Model of a File System....64
3.2.1 Fundamental System Components....64
3.2.1.1 Storage Blocks....65
3.2.1.2 Information Linkages....67
3.2.2 Feasible Storage Blocks....69
3.3 Single Device File Organization....70
3.3.1 Purpose of Analysis....70
3.3.2 Costs....70
3.3.3 Assumptions....71
3.3.4 Analysis of Alternative Structure Concepts....72
3.3.4.1 Unfeasible Structures....72
3.3.4.2 Feasible Structures....75
3.3.4.3 Index File Segmentation Based on Index Record Length....76
3.3.5 Evaluation of Mixed Structure....81
3.3.5.1 Case 1: N(M)<n-N(M). Largest Group of Keys Less Than 50% of File....81
3.3.5.2 Case 2: N(M)<n-N(M). Largest Group of Keys Greater Than 50% of File....82
3.3.6 Segmentation of Index Files Based on Key Length....83
3.3.6.1 Objective of Analysis....84
3.3.6.2 Storage Minimization....84
3.3.6.3 Processing Cost Minimization....85
3.3.6.3.1 Design of Optimization Procedure....86
3.3.6.3.2 Algorithm for Search Processing Cost/Total Processing Time Optimization....86
3.3.6.3.3 Parameters....87
3.3.7 Segmentation of Index Files Based on Usage Frequency....94
3.4 Consideration of Request Utility as a Function of Average Retrieval Time....96

APPENDIX: PARAMETER DEFINITIONS FOR FILE STRUCTURE ANALYSIS.....98

TABLE OF CONTENTS (Cont.)

Page

4. THE ANALYSIS OF BIBLIOGRAPHIC FILE STRUCTURES, USING INDEXED SEQUENTIAL ORGANIZATION by Jorge Hinojosa. 101

- 4.1 Introduction 101
  - 4.1.1 Object 101
  - 4.1.2 Cost Relations 101
  - 4.1.3 The Problem 103
- 4.2 General System Structure 103
  - 4.2.1 File Structures Selected for Analysis 103
    - 4.2.1.1 Access Record Mapping 103
    - 4.2.1.2 Connection with Previous Analysis ... 105
    - 4.2.1.3 Record Format 108
  - 4.2.2 Comparison of Structures 5 and 7 108
- 4.3 Division of an Index File Based on Key Lengths 109
  - 4.3.1 Cost of Storage 110
  - 4.3.2 Cost of Time 111
  - 4.3.3 Dynamic Programming Formulation 112
  - 4.3.4 Solution of the Problem 115
  - 4.3.5 Sample Computations and Results 115
- 4.4 Division of an Index File Based on Usage Frequency .. 117
  - 4.4.1 Assumptions 117
  - 4.4.2 Condition for Feasibility 117
  - 4.4.3 Required Distribution 119
  - 4.4.4 Growing Rate of Indexes 120
    - 4.4.4.1 Small Files 120
    - 4.4.4.2 Large Files 121
      - 4.4.4.2.1 Case A 121
      - 4.4.4.2.2 Case B 121
  - 4.4.5 Conclusions 123
  - 4.4.6 An Extension 124
- 4.5 Estimation of Overflow Areas for Index Files 124
  - 4.5.1 Overflow Records 124
  - 4.5.2 Probabilistic Model 124
    - 4.5.2.1 New Key Expected Insertion 125
    - 4.5.2.2 Estimation of Overflow Areas 125
  - 4.5.3 Alternative Estimation 126
- 4.6 Allocation of Non-Keyed Records to Storage Blocks ... 127
  - 4.6.1 Mathematical Model 127
  - 4.6.2 Inefficiency of Random Allocation 129
  - 4.6.3 Computational Tool 129
- 4.7 Estimation of Overflow Areas for the Pointers File .. 130
  - 4.7.1 Former Scheme 130
  - 4.7.2 New Scheme 130
  - 4.7.3 Differences 131
  - 4.7.4 Stochastic Model 131
    - 4.7.4.1 Assumptions 131
    - 4.7.4.2 Proof of Stationarity 132
    - 4.7.4.3 Distribution of Xij 133
  - 4.7.5 Estimation of Overflow Space 135

TABLE OF CONTENTS (Cont.)

Page

4.8 Multiple Device Systems....136
  4.8.1 General....136
  4.8.2 DASD Control Units....136
    4.8.2.1 Device Combinations....136
    4.8.2.2 Device Selection....136
  4.8.3 Cost Comparisons....137
  4.8.4 Vertical Division of a File....137
    4.8.4.1 Usage Frequency Distributions....140
    4.8.4.2 Cost Equations....141
    4.8.4.3 Equivalent Problems....142
    4.8.4.4 Required Cumulative Distributions....142
    4.8.4.5 Program DUODEV....147
    4.8.4.6 Some Results....147
    4.8.4.7 An Extension....148
  4.8.5 Conclusions....151

5. FIRST-STAGE MODEL OF THE ECONOMIC EFFECTS OF INCORPORATING A DATA COMPRESSION SYSTEM INTO AN ON-LINE DIRECT-ACCESS STORAGE AND RETRIEVAL SYSTEM by Kelley L. Cartwright 155

5.1 Introduction: Compression and Coding....155
  5.1.1 The Advantages and Constraints of Compression....155
  5.1.2 Coding Procedure....156
5.2 A Model of the Effect Upon System Costs of Incorporating a Huffman Coding Scheme into a Storage and Retrieval System.157
  5.2.1 General Effect Upon the System....157
  5.2.2 General Evaluation Criteria....158
  5.2.3 General Model....158
  5.2.4 Analysis of Variables....159
    5.2.4.1 Step 3: Machine Processing Prior to Storage....159
    5.2.4.2 Step 4: Storage on Disc....163
    5.2.4.3 Amortization of Costs of Steps 3 and 4 (a)....164
    5.2.4.4 Step 6: Retrieval of Citations....164
    5.2.4.5 Step 7: Processing of Citations for Output....164
  5.2.5 Huffman Codes....167

6. IMPLEMENTATION OF BIBLIOGRAPHIC RECORD COMPRESSION by Vikas Sahasrabudhe and Ashok Kulkarni 171

6.1 Overview....171
6.2 Record Encoding....171
6.3 Table Structure....172
  6.3.1 First Letter Tables....172
  6.3.2 Second Letter Tables....174
  6.3.3 Encoding Example....176
6.4 Record Decoding....181

TABLE OF CONTENTS (Cont.)

Page

6.4.1 Decoding Table Structure....181
6.4.2 Decoding Example....182
6.5 Frequency Analysis and Alphabet Selection....185
6.5.1 Extension to Digrams....185
6.5.2 Extension to k-Grams....187
6.6 IBM 360 Implementation....187
6.6.1 Analysis Program....188
6.6.1.1 COUNT (TABLE1, TABLES, NUMBER, NARC, MARKER, TAB2NM, TOTAL)....189
6.6.1.2 ADJUST....194
6.6.1.3 HUFTRE....197
6.6.1.4 HUFCOD....199
6.6.1.5 IODISK (TABLE1, TABLES, TAB, TREE)....201
6.6.2 ENCODE....201
6.6.3 DECODE....202
6.6.4 Extension of Program for Inclusion of k-Grams (k>3)....205
6.6.5 Program Operation and Results....206
6.7 Results....208
6.7.1 Initial Method....208
6.7.2 Second Method....208
6.7.3 Third Method....208
6.7.4 Effect of Varying Source Alphabet Size....209

7. SPECIFICATION FOR FORMAT TRANSLATION OF THE SANTA CRUZ FILE by Jay L. Cunningham 213

7.1 Introduction....213
7.1.1 Translation from Santa Cruz Original Format to ILR Input....216
7.1.2 Translation from ILR Input to ILR Processing Format....216
7.1.3 Translation from ILR Processing Format to (Latest) MARC II....216
7.2 Translation from Santa Cruz Original Format to ILR Input Format....217
7.2.1 Condition of Source File....217
7.2.2 Format Translation Tables to Achieve ILR Input Format....217
7.2.3 Notes to Figure 67....217

8. CRUNCH: THE SANTA CRUZ FILE TRANSLATION SYSTEM by John M. Reinke 235

8.1 Introduction....235
8.1.1 Main Program: LAYOUT1....235
8.1.2 Subprogram: STRINGER....235
8.2 Subroutine ES200500....236
8.3 Subroutine ADDALG....236
8.4 Subroutine BIBLIO....237
8.5 Subroutine PPD....237
8.6 Subroutine PETSAUTH....239
8.7 Subroutine TITEALG....239

TABLE OF CONTENTS (Cont.)

Page

9. PROSPECTS FOR AUTOMATIC FIELD RECOGNITION by John M. Reinke 249

9.1 Introduction....249
9.2 An Example of the Proposed Use of Pre-scanning in Algorithm Design....250
9.3 A Proposed System for AFR Algorithm Development....252
9.4 Advantages of the AFR Algorithm Development (AFRAD) System....254
9.5 Remarks on Sampling....255
9.6 Pre-scanning and Mapping....256
9.7 Automatic Language Recognition....257
    9.7.1 Subroutine #1....257
    9.7.2 Subroutine #2....258

10. TRANSLATION FROM ILR PROCESSING FORMAT TO MARC II COMMUNICATIONS FORMAT by Jay L. Cunningham. 265

10.1 Introduction....265
    10.1.1 Purpose....265
    10.1.2 Scope....265
    10.1.3 Contents....265
10.2 Narrative Description of Overall Program Structure....265
    10.2.1 Housekeeping....265
    10.2.2 Read the Next INFOCAL Record to an Input Buffer....265
    10.2.3 Shredout of INFOCAL Record Directory....265
        10.2.3.1 Major Loop....265
        10.2.3.2 Bookkeeping....266
    10.2.4 Major Loop....266
    10.2.5 Create Fixed Field 008....266
    10.2.6 Create New Leader....266
    10.2.7 Create New Directory....267
    10.2.8 Write the Completed Record to Output Storage....267
    10.2.9 End of File Check....267
10.3 Flowcharts....270
10.4 Summary of MARC II Record Structure (Communication Format)....280
    10.4.1 Modifications to Overall Record Structure to Produce a Revised MARC II Record....280
        10.4.1.1 Field Terminators....280
        10.4.1.2 Subfield Delimiter Codes....281
        10.4.1.3 MARC II Indicators 1 and 2....281
    10.4.2 Modifications to Leader....282
        10.4.2.1 Structure and Content of MARC II Leader....282
        10.4.2.2 Changes in INFOCAL Leader - Translation Table....283
    10.4.3 Modifications to Directory....285
        10.4.3.1 Structure and Content of MARC II Directory....285
        10.4.3.2 Changes in INFOCAL Directory....286
    10.4.4 Modifications to Fixed Length Data Elements Field...291

TABLE OF CONTENTS (Cont.)

Page

10.4.4.1 Structure and Content of MARC II Tag 008 Field....291
10.4.4.2 Changes in INFOCAL Fixed Length Data Element Field - Translation Table....293
10.4.5 Modifications to Variable Fields....295
10.5 Specifications for Building a Santa Cruz - UC MARC File....298
10.5.1 Overall Record Structure....298
10.5.2 Leader....301
10.5.3 Fixed Length Data Elements Field - Tag 007 (UC MARC)....301
10.5.4 Variable Fields - Redefinition of Tag 090 (Local Headings)....302

LIST OF FIGURES

Figure Title Page

CHAPTER 1: AN ANALYSIS OF THE SEARCH PROBLEM IN FILES OF OBJECT DESCRIPTIONS

1. One Value Associated with Elements from Two Different Sets of Entities 4

2. Two-by-Two Contingency Table 12

3. Functional Similarity. 18

4. Elemental Similarity 19

CHAPTER 2: THE ASSIGNMENT OF INDEX TERMS

5. Distribution of Number of Applications (Tokens) of Unique Headings (Types) 28

6. Distribution of Documents by Number of Subject Headings Assigned 31

7. Characteristics of the File. 32

8. Number of Documents Indexed and Subject Headings Introduced, 1880-1967. 34

9. Blow-Up of Fig. 8 with Mean Number of Subject Headings Assigned/Doc and Number of Documents and Subject Headings (1912-1948). 36

10. Mean Co-Assignment for Differences between Mean Date and Date of Individual Application 38

11. Mean Co-Assignment for Differences between Earliest Date and Date of Individual Application (Sample Base ≥20 Only). 41

CHAPTER 3: DESIGN OF FILE STRUCTURES FOR ON-LINE BIBLIOGRAPHIC CONTROL SYSTEMS

12. Types of Storage Blocks (SB's) 67

13. Types of Information Linkages. 68

14. Unfeasible Blocks. 69

15. Processing Cost Curve. 70

LIST OF FIGURES (Cont.)

Figure Title Page

16. Unfeasible Structures 74

17. Feasible Structures 77

18. Distribution of the Number of Records Associated to Each Key 79

19. Structure 6 ("Mixed") 80

20. Structure 7 ("Modified Mixed") 80

21. Distribution of Key and Record Lengths 84

22. Matrix for *K* 90

23. Total Cost/Total Processing Time 93

24. Usage Frequency Distribution 94

25. Average Request $/Average Retrieval Time 96

26. All Requests $/T.P.T.. 97

CHAPTER 4: THE ANALYSIS OF BIBLIOGRAPHIC FILE STRUCTURES, USING INDEXED SEQUENTIAL ORGANIZATION

27. Operational Cost of a File 102

28A. Mapping of Master Records into Access Records for Structure I. 104

28B. Mapping for Structure II 104

29A. Feasible Structures: Structure 5a - Fixed-Blocked Unit Lists. 105

29B. Structure 7 - "Modified Mixed" 105

30. Structure 7 - "Modified Mixed" with Two-Level Lists. 107

31. Key Length Distribution. 108

32. Pointers Distribution. 108

33. Costs for a Divided File as a Function of the Set of Division Points (Di) 111

LIST OF FIGURES (Cont.)

Figure Title Page

34. Flow Chart for Dynamic Programming Computations. 114

35. Parameters and Cost Values Used. 115

36. Cost per Request as a Function of the Ratio System Volume-File Size for Five Different Division Strategies 118

37. Required Cumulative Distribution 119

38. Cumulative Required Distribution - Case A. 122

39. Network Representation 128

40. Representation of Transition Probabilities 134

41. Total Search Cost per Request as a Function of the Ratio System Volume-File Size, for Different Devices 138

42. Total Search Cost per Request as a Function of the Ratio System Volume-File Size, for Different Devices 139

43. Usage Frequency Distributions. 140

44. Normalized Usage Frequency Distribution. 141

45. Cumulative-Required Distribution for a Two-Device File (2314-2321) 143

46. Cumulative-Required Distribution for a Two-Device File (2314-2321) 144

47. Cumulative-Required Distribution for a Two-Device File (2314-2321) 145

48. Artificial Cumulative Usage Frequency Distributions. 146

CHAPTER 5: FIRST-STAGE MODEL OF THE ECONOMIC EFFECTS OF INCORPORATING A DATA COMPRESSION SYSTEM INTO AN ON-LINE DIRECT-ACCESS STORAGE AND RETRIEVAL SYSTEM

49. Index of Variable Names. 161

50. Combined Model 166

51. Reduction of a Set of Probabilities. 168

52. Creation of a Compact Code 168

LIST OF FIGURES (Cont.)

Figure | Title | Page

53. Effect of Character Frequency on Number of Bits of Compact Variable-Length Code for 6-Character Alphabet 169

54. Tree Structure for Computing Huffman Code Lengths 170

CHAPTER 6: IMPLEMENTATION OF BIBLIOGRAPHIC RECORD COMPRESSION

55. Examples of Termination Entries for Illustrative Huffman Codes 173

56. List of Digrams for the Encoding Example. 176

57. Example of First- and Second-Letter Tables. 178

58. Conversion Table. 179

59. Probabilities and Huffman Codes of Characters c_1, c_2, c_3, c_1c_2, c_2c_3, and c_4. 183

60. Data Analysis CSECT Function and Parameters 190

61. Layout of Count Tables. 192

62. Structure of Tree Array 197

63. Encoding Process. 203

64. Compression Results 209

65. Huffman Coding Procedure Results. 212

CHAPTER 7: SPECIFICATION FOR FORMAT TRANSLATION OF THE SANTA CRUZ FILE

66. Santa Cruz File Translation Run 215

67. Format Translation Table I-Field Segment. 218

68. Fixed-Length Publication Dates Logic. 223

69. Logic for Type of Added Entry Code - Series 225

70. Format Translation Table A-Field Segment. 226

71. Data Element Patterns and Associated Coding Imprint Field . . . 230

72. Format Translation Table B-Field Segment. 231

LIST OF FIGURES (Cont.)

Figure Title Page

73. Disposition of Shelf Key Data CT 000 - Shelf Key Card, Santa Cruz. 233

CHAPTER 8: CRUNCH: THE SANTA CRUZ FILE TRANSLATION SYSTEM

74. Flowchart of Subroutine ES200500. 241

75. Flowchart of Internal Subroutine ADDALG 242

76. Flowchart of Internal Subroutine BINLIO 243

77. Subroutine PPD Flowchart. 244

78. Flowchart of Subroutine PERSAUTH. 247

79. Subroutine TITLEALG Flowchart 248

CHAPTER 9: PROSPECTS FOR AUTOMATIC FIELD RECOGNITION

80. Unique Diatable Displacement Values 260

81. Short Words Commonly Found in Several Foreign Languages 261

82. Language Recognition Algorithm #2 Flowchart 263

CHAPTER 10: TRANSLATION FROM ILR PROCESSING FORMAT TO MARC II COMMUNICATIONS FORMAT

83. Tag Translation Table 268

84. Directory Re-building Control Table 269

85. Format Translation Table Leader Segment 283

86. Indicator Revision. 286

87. Infocal Directory Entry Structure 287

88. Reorganization of Indicator Values. 288

89. Format Translation Table Fixed Length Data Elements Segment 293

90. Variable Field Tag Translation Table. 299

LIST OF TABLES

Table　　Title　　Page

CHAPTER 2: THE ASSIGNMENT OF INDEX TERMS

1. Rate of Addition of New Terms: Date and Mean Headings per Document 51

2. Rate of Addition of New Terms: Mean Publication Date and Earliest Date of Appearance 53

3. Relative Age of Subject Fields 55

4. Major Hypothesis Data: Mean Date of Publication 57

5. Major Hypothesis Data: Earliest Date of Publication 59

CHAPTER 4: THE ANALYSIS OF BIBLIOGRAPHIC FILE STRUCTURES, USING INDEXED SEQUENTIAL ORGANIZATION

6. Minimum Total Search Cost per Request (Dollars) 116

7. Percent of Records in Each Subfile 116

8. Cumulative Required Distribution, Case B 123

9. Comparison of Search Costs for Two Systems 148

10. Total Search Cost as a Function of System Volume for a Two-Device File 149

11. Total Search Cost per Request as a Function of System Volume for a Two-Device File 150

12A. Total Search Cost per Request for a Two-Device File, Based on Usage Frequency Distribution and Key-Length Distribution . . . 153

12B. Comparison of Total Search Cost per Request, for Three Basic Structures 154

CHAPTER 6: IMPLEMENTATION OF BIBLIOGRAPHIC RECORD COMPRESSION

13. Results for Huffman-coding Procedure for Characters and Digrams in 200 LC MARC II Records 211

FOREWORD

This report contains the results of the second phase (July, 1968 - June, 1970) of the File Organization Project, directed toward the development of a facility in which the many issues relating to the organization and search of bibliographic records in on-line computer environments could be studied. This work was supported by a grant (OEG-1-7-071083-5068) from the Bureau of Research of the Office of Education, U.S. Department of Health, Education, and Welfare and also by the University of California. The principal investigator was M.E. Maron, Professor of Librarianship and Associate Director, Institute of Library Research; the project director and project manager were, respectively, Ralph M. Shoffner and Allan J. Humphrey, Institute of Library Research.

This report is being issued as seven separate volumes:

- Shoffner, Ralph M., Jay L. Cunningham, and Allan J. Humphrey. The Organization and Search of Bibliographic Records in On-Line Computer Systems: Project Summary.
- Shoffner, Ralph M. and Jay L. Cunningham, eds. The Organization and Search of Bibliographic Records: Component Studies.
- Aiyer, Arjun K. The CIMARON System: Modular Programs for the Organization and Search of Large Files.
- Silver, Steven S. INTX: Interactive Assembler Language Interpreter Users' Manual.
- Silver, Steven S. FMS: Users' Guide to the Format Manipulation System for Natural Language Documents.
- Silver, Steven S. and Joseph C. Meredith. DISCUS Interactive System Users' Manual.
- Smith, Stephen F. and William Harrelson. TMS: A Terminal Monitor System for Information Processing.

Because of the joint support provided by the Information Processing Laboratory Project (OEG-1-7-071085-4286) for the development of DISCUS and of TMS, the volumes concerned with these programs are included as part of the final report for both projects. Also, the CIMARON system (which was fully supported by the File Organization Project) has been incorporated into the Laboratory operation and therefore, in order to provide a balanced view of the total facility obtained, the volume is included as part of the Laboratory project report. (See Maron, M.E. and Don Sherman, et al., An Information Processing Laboratory for Education and Research in Library Science: Phase 2. Institute of Library Research, 1971.)

ACKNOWLEDGEMENTS

This volume contains the results of Institute work dealing specifically with issues and problems of organizing and accessing large bibliographic files. For the most part, each chapter was written by the staff member actually conducting the study. We gratefully acknowledge Irene Travis, Marcia Bates, Jorge Rodriguez, Jorge Hinojosa, Kelley Cartwright, Vikas Sahasraboudhe, Ashok Kulkarni, Jay Cunningham, and John Reinke.

In addition to the authors, many Institute staff members contributed their special abilities toward completion of the File Organization Project. Because the study itself was broad in scope, there were varied tasks requiring the talents of a wide spectrum of people. In particular, much of our work involved considerable computer programming, without which these particular studies could not have been accomplished. Specifically, we wish to thank for their design and programming efforts Arjun Aiyer, Dennis Fried, Bill Harrelson, Birdie Hodges, Steve Johnson, Naresh Kripalani, John Reinke, and Steve Smith.

Finally, we wish to thank and to commend the work of those who worked with special zest in the preparation of these pages: Carole Fender, Linda Herold, Scott Herold, Jan Kumataka, Pat Oyama, and Rhozalyn Perkins.

1. AN ANALYSIS OF THE SEARCH PROBLEM IN FILES OF OBJECT DESCRIPTIONS

By Irene Travis

1.1 The Purpose and Scope of This Chapter

This chapter is intended as an initial exploration of "the search problem" for the File Organization Project of the Institute of Library Research at the University of California at Berkeley. To avoid any misapprehension of its scope and purpose, it seems appropriate to clarify immediately what is meant here by "the search problem." As a topic in the literature of information retrieval, "the search problem" refers, in fact, to a variety of topics. Two of these will now be specifically excluded from consideration. The first is the common definition of "the search problem" as minimizing the time required to find an item in a machine store. The other topic excluded is "the search problem" as that of finding "relevant" documents, the point of view of the literature searcher. The focus will be instead, on record retrieval, on the problem of retrieving records corresponding to a searcher's query.

Bibliographic files will be used as examples, but the analysis extends to a considerably larger class of systems, as will be clear in the course of the discussion. The aim of this chapter is to obtain an overview of the problems of searching files like bibliographic files without limiting attention to any one such set of records. In the course of this discussion, the construction of files, the problems of searching them, and the nature of the tools which a system might incorporate to aid in file search will be examined.

1.2 A General View of Object Description Files

1.2.1 Data Structure and Data Processing

The way in which man stores his knowledge of the world in his brain and nervous system, processes it, and converts the results to a form suitable for communication with other men, is little known at present. Since man learned to record his communications in more or less permanent media, however, certain types of information which we use have come more and more to be stored externally; that is, stored in graphic, written, sound-recorded, or other forms. Indeed, the amount of information about the world which is now accumulated is far too vast and particular for more than a minute part of it to be held in human memories at any one moment.

Information, whether in the brain or in some external form, is used and modified in many ways for many purposes. The type of processing possible is clearly related to the nature of the information and the way in which it is structured, but our knowledge of these matters is as yet quite limited. The most complex forms of storage and processing are still those confined to the human brain, despite the fact that for certain classes of data and processes, the electronic computer is a far faster and more reliable tool. In other words, the brain is still the only means we have for storing certain data in ways such that certain classes of processes can be performed on them. The coalescing and transformation of data involved in determining the meaning of a sentence is an example of such a class of processes. Writings in natural language are processable only in very limited ways, without transformation, either by machine or by the brain. Each must transform it to other data structures before complex processing can be done. In recent years as the amount of information has increased, problems associated with organizing these external stores in a fashion useful for machine processing, that is, of discovering methods for the external processing of the external stores, have received even more attention.[1,2]

If we look at the kinds of data which have been successfully used by machines to this point, we find that much of it can be characterized as descriptions of objects or "entities": persons, buildings, organizations, filing cabinets, books, etc. A large sub-class of such data are those which are observations of entities involved in systems which can be adequately represented by some mathematical or statistical model, such as the structural aspect of a building. The rules for manipulating such quantitative data are "formal": they are independent of the particular datum. The machine can manipulate these data not only as well as the human brain but with a speed and accuracy many times greater than any person. Those descriptions, on the other hand, which are not observations from a system which can be adequately represented formally are frequently a mixture of strings of words and some measurements. The kinds of data contained in personnel records is a good example. It is this latter type of data with which we will be concerned.

Apart from updating, matching, sorting, and counting of records are the most usual types of processing carried out on such descriptions. Its purpose is usually one or a combination of the following:

a. Finding information about individual objects;

b. Allowing the generation of quantitative data about the size of various classes of objects as a means of characterizing the file or the group of objects described;

c. Creating a directory to the objects described.

Few of such data retrieval systems are used for any one of these purposes exclusively.

The processes which support even these purposes in our present systems, although very fundamental and very useful, are still quite limited. Such determinations as, for example, *set membership* using non-numeric data may be difficult. Membership frequently cannot be decided by any of the order relations such as "less than" or "greater than" which are very powerful for numeric data, nor, sometimes, will the list of possible members or non-members be enumerable and storable. An example of such a difficult determination is the recognition of personal names; another, the determination of the language of a string. This type of knowledge does not have to be explicitly stored for the human user of the library catalog, for instance, who recognizes such distinctions with little difficulty, but the machine algorithms for recognizing these accurately are as yet rudimentary and troublesome to develop. In truth, the manner in which humans make these inferences is not formally understood, and the development of data structures for machines to allow the simulation of such capabilities is still in its infancy. Our present programs can use only a limited but important class of formal clues, such as the punctuation of the string or the occurrence of certain symbols of words, although this approach is sufficient for certain applications. It is the search problem in files of observations of this class of largely non-quantified object descriptions which we will now explore in this paper.

1.2.2 Object Descriptions - Entities, Attributes, and Values

1.2.2.1 The Domain of Discourse

The domain of discourse of a file is that set of objects or entities which is described in that file. These descriptions are the values ascertained for certain properties or attributes of these entities. For example, if the objects are people, we might wish the values of attributes "height," "weight," "vision" or similar characteristics; in other cases we might be more concerned with attributes such as "highest academic degree" or "number of years experience as a supervisor." These attributes would have values such as "150 lbs." or "20-40," or "M.A." *Values, then, are the data in the file*.

These values are each associated with an element of the domain of discourse; however, given these data or, in other words, simply looking at a previously prepared description, there are several alternative ways of associating these values with entities. In other words, there is usually more than one possible kind of object in domain of discourse of the file. The reason these alternative analyses can exist is that frequently *the value of one attribute of an entity may itself refer to another entity*. For example, monographs have an attribute "author" which frequently has as a value a name of a person. The person, however, is also an entity with an attribute "name" whose value is the same as that of "author." (See Fig. 1)

The discussion of the domain of discourse presented in "Analytic Information Retrieval" by L.E. Travis^3 bears on this point. If a file is a set of descriptions of some homogeneous class of objects, as is a file used primarily as a directory to that class of objects, L.E. Travis calls this class the "primary members of the domain of discourse." The other entities to which the data also refer are termed the "secondary members." The three types of file use which might lead the system designer to consider these secondary members and their relations to one another and to the primary members are roughly parallel to the three general uses described above. They are:

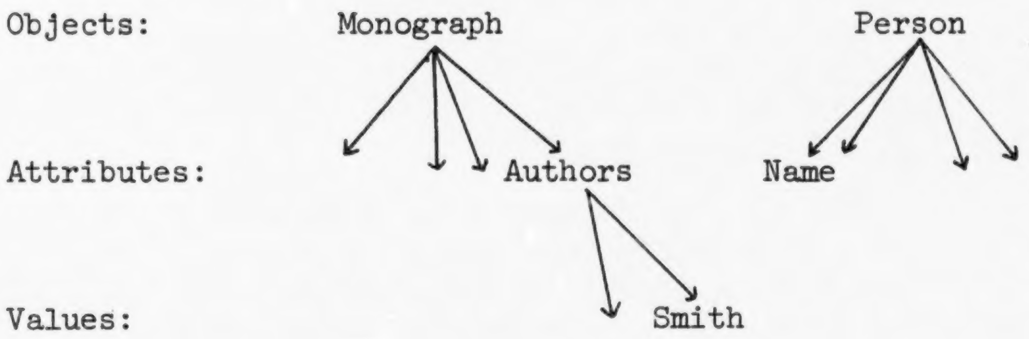
a. Retrieving data about the secondary entities themselves, e.g., finding the death date of an author from the catalog entry of one of his books.

b. Counting the number of members of various sub-groups in order to characterize a set of secondary members of the domain of discourse. E.g., how many different publishers are represented in the catalog of the University of California library? Has the distribution of publishers altered radically within the last ten years?

c. Utilizing information about the secondary entities to discover which primary member in the directory might be of use.

FIG. 1:

ONE VALUE ASSOCIATED WITH ELEMENTS FROM TWO DIFFERENT SETS OF ENTITIES



In deciding the number and types of access points to a file, the designer must therefore consider the importance of each of the attributes of the secondary members as search clues for the primary members as well as user interest in the secondary members themselves. The other criteria are, of course, the cost and storage constraints of the system. At one extreme there may be only one type of access for the whole file, such as "member number"; at the other, each attribute might be a possible access point.

L.E. Travis speaks of the sets of primary and secondary members as "relatively homogeneous classes" (Travis, p. 317), and we will now examine the influence of this property on file structure. To explore the matter, let us first take a file consisting of object descriptions corresponding to a single set of primary members in the domain of discourse. The secondary members, if any, are by definition mentioned only as values of the attributes of the primary members. Their selection is not direct as a result, nor are the listed attributes selected for the information they contribute about the secondary members. Their mention in the file is, in a certain sense, accidental. The primary members, on the other hand, are selected according to some classifying criteria. It is with regard to these criteria that the domain is homogeneous. For example, in a hospital file the selecting characteristics might be that the entities are (1) people who were (2) patients in the hospital (3) since 1967. An important thing to note, however, is that in such a file, those attributes which define the class are usually not recorded in the object description in the file since they carry no information. They will, on the contrary, be assumed by the searcher, and they do not distinguish the objects represented in the file. The object description, then, which occurs in the file is not the characterization of the class, but, rather, given the characterization, the attributes used in the description constitute a framework for gathering and recording data.

1.2.2.2 The Attribute List

Since the concept of the class is prior to the selection of the attributes for describing the objects, it, of course, plays an important role in their selection. In the first place, due to the great value of standardization to searching any but very small files, we do not usually think of selecting the attributes for each entity separately. Rather, we conceive of a set of attributes which "holds" for the class where "holds" may be interpreted as follows:

> Given any set of defining characteristics for the class and any proposed attribute, that attribute holds for the class if the null value of the attribute is informative; that is, if the null value could be ascertained only by examining the individual member and is not inferrable from the class definition.

To rephrase this definition,

> An attribute holds for a class, if, given the class definition, the a priori probability that it holds for each individual is greater than zero.

For example, one may include the class attribute "number of grades completed" in a file of "people"; although they may not all have attended school, we cannot ascertain that fact only by our knowledge of what it is to be a "person." The null value, then, or, in other words, the failure of the attribute to hold for the individual, is informative.

1.2.2.3 Attributes and Record Formats

To pursue the matter on a less abstract level, this list of class attributes is ordinarily translated into the actual file structure by the rules constituting the record format. With such a list, it is possible to put explicitly in the record only the values of the attributes while the attributes themselves are carried implicitly either by relative location in a fixed format record or by tags in a more flexible structure. We shall call the area of the record where the value of an attribute is stored, when it occurs, its attribute field. In terms of file structure, then, the degree of "homogeneity" of the domain of discourse is the extent to which it is efficient to describe all objects in the domain using one format.

To look now at a more complex situation, if one has more than one file of object descriptions in the system, the dichotomy between the class concept and the description still exists, but the class concept can no longer be assumed. It will be reflected in the routine for selecting files and adopting the correct format in order to channel search requests to the proper file and attribute field.

The nature of formats as a tool for structuring data is now apparent. If our application is such that (1) our files are object descriptions only and (2) that the a priori classification of these objects is not objectionable, formats are a device for compressing and standardizing files. It is a powerful tool precisely because there are a multitude of applications in which that data meet these two requirements. The second one, after all, really only implies that we are interested in well-defined internal distinctions and similarities within one particular set of objects rather than in the relations of these individuals to entities outside the class. These internal distinctions and similarities are determined by examination and manipulation of the values associated with each attribute; therefore, the nature of these "value sets" and their role in the file structure will next be considered.

1.2.2.4 Value Sets and Record Fields

The process of obtaining the values which appear as data in the file has two stages: observation and recording. Each has rules and procedures governing its performance. Although these rules may not always be layed out consciously, in complex systems such as library catalogs they are contained in extensive and detailed codes. The reliability of the descriptions as guides to the objects they represent is, of course, dependent upon the quality of the rules and the consistency with which they are applied. Our focus on file organization excludes the former problem, but we shall be interested in whether or not the rules exist explicitly, what they are, and, of course, whether they were consistently applied in constructing the file. All this information has direct bearing on the problems of record retrieval.

The devices used for the observation of the values, to consider the first step, can range in complexity from a single glance to the use of sophisticated mechanical, electronic, or statistical tools. The rules may range in exactness from the highly objective to the almost completely subjective (with resulting inconsistencies).

When these observations are recorded, they frequently undergo transformations. These changes are of two kinds: first, formal structuring, such as punctuation or transliteration of foreign alphabets; secondly, conversion to "legal" values; that is, there may be a limited list of values of color, for example, which may be used to describe the objects, and the describer must "reduce" each observation to one of these values. In mathematical terms, the change is a many-one mapping. A mapping takes one set of values called the "domain" and supplies a rule or "function" for transforming them to another set of values called the "range" of the mapping. In all many-one mappings, the original - in this case, the observed - values are not recoverable from the transformed, or recorded, values. A simple example of a "recording function" is the following rule:

"Round-off 'annual income' to the nearest $100."

This rule defines the transformation. The range is given indirectly; that is, a rule for calculating it is given. The range consists of all integers whose last two digits are '00'. If, moreover, the data gatherer knows "annual income" to the penny, there will be 10,000 possible observed values for every "legal" one. Here the rule is explicit, but the range must be calculated. In contrast, in library subject cataloging the range is explicit in the subject heading list, but, unfortunately, the conversion and observation rules are totally unformalized.

The set of recorded values associated with a single class attribute, as, for example, a list of all the authors which occur in the file, will be called a value set. Associated with the set of types (the value set) is a set of occurrences or tokens. Just as the class attributes correspond to attribute fields in the file, this latter set may be regarded as being in one-to-one correspondence with the actual data strings in the fields of the logical records. The mapping between these two sets defines the final stage in the process going from the "true" values to their representation in the record: that is, from the "true" values to the observed values; from the observed values to the set of occurrences or token; and finally from the set of tokens to a specific location in the record itself. One goes in the last step from, say, the fifth occurrence of the twentieth value of the tenth attribute in the set of objects being described to a record location; for example, the second repeat of the fifth attribute field of the 410th logical record.

We have now finished describing the creation of "object-description files." We have approached it essentially as a process of selecting and mapping data about the real world into a file and record structure, delineating the many decision points involved. This analysis has prepared the way for a discussion of the problems of searching such files. We will concentrate on the problems raised by the use of such files as directories to the objects described, since this case presents all the problems raised by other uses and more.

1.3 Searching Directories

1.3.1 The Query for a Directory Search

There are close parallels between the process of constructing the file of object descriptions and the process of using the file as a directory; indeed, our interest in the former in a paper on file search stems from this similarity. The primary distinction is that the searcher is in a state of uncertainty about the file and its contents, in addition to any uncertainty he may have about what sort of objects might fulfill his need. Our interest will begin at the point at which the user's request is to be translated into a search of a particular directory. This search may be hampered by either lack of knowledge about the rules and content of the file or uncertainty about the class of objects which is being searched for. This latter difficulty may stem either from lack of adequate observation of the class of objects sought or because the class is only hypothetical, as in the set of books about the psychological environment of public libraries. In the first case the class is known to exist; indeed, the searcher may even have observed its members. The problem is to find the descriptions of the objects in this particular file in order to ascertain their inclusion in it, their location, availability, or suitability to the searcher's need. If the class is hypothetical, on the other hand, the searcher must first ascertain the existence of such a class and its membership, if any, or, worse problem, ascertain its *nonexistence*. In any case, however, the searcher must select a domain, a set of attributes and the set of values he thinks might most likely have been used to describe the set of objects he seeks. He must further select them in such a way that they correspond to the file-constructor's practice at each of these points if the search is to be successful. The system, of course, may provide some help by supplying him with the system construction rules and certain error-correcting routines. Clearly, the more explicit and better defined the rules, the more help they will be to the user. The types of aids which the system may provide and their role will be considered next as we turn from the uncertainties which cause problems in file search to the sources of information which can aid the searcher or the system in making correct matches.

1.3.2 Retrieval Failures

Corresponding to the three sources of uncertainty mentioned above, (1) lack of knowledge about the files rules, (2) lack of knowledge about the object or class of objects the searcher is describing, and (3) lack of knowledge about the actual contents of the file, we can describe three classes of system or "retrieval" failures which we will call *formal errors*, *variants*, and *content errors*. Let us first consider those retrieval failures caused by *formal errors*. They are those differences between the query and values in the record against which it is to be matched which result from one or both forms being incorrect with respect to the system's construction rules or "common knowledge" or language such as is available in a dictionary, itself a set of rules. The importance of this class is that these errors can be identified and corrected *without knowledge of the actual contents of the file or further observation of the objects*. They are *formal errors* in the sense that *knowledge of the rules alone is sufficient to correct them*. These include, specifically, such mistakes as failure to put the "main entry" in specified form in a library catalog, spelling errors in common words, typographical errors which a user or proofreader could reasonably correct from his own knowledge, misplaced parentheses in Boolean search queries and other similar faults.

The second and third class of retrieval failures, those caused by content errors and variants, by contrast, can only be detected through knowledge of the contents of the file itself or by tracing back to other accurate descriptions of the objects, if they are available. *Descriptions by the system and by the searcher or any two describers which are completely "correct," but which because of ambiguities in the rules or inexactness in the methods of observation or differences in completeness of information fail to match are here called "variants."* An example of a variant is the use of a man's full middle name in a query and his initial only in the file. These can be detected only by examining the file itself, not by examining the objects or the rules. Content errors are mistakes in the contents of the query (or the file) which result from faulty information about the object, on the part of either the system or the user. Content errors include such things as a spelling of a name in a query which is phonetically reasonable but not the actual spelling; an inexact title (of a document); or an incorrect date which is not obviously absurd. In theory, one could frequently trace this faulty information back to a correct source other than the file, but in practice that procedure would often be impossible, and, moreover, as the error would not be detected in the search process unless the correct record were in fact retrieved, the cause of the failure might not be recognized. The significance of variants and content error is that, if they are to be detected and corrected at all, the system must do it internally as part of the matching process; unlike formal errors, these mistakes cannot be corrected practically from any other source.

From this analysis of the possible sources of retrieval failures, it follows that the criteria for deciding whether or not to supply search aids is different depending on whether the failure is due to formal error, content error, or variant. In the first case the criterion is the level of participation which the system wishes to require of its users; the less the demand, the more "formal error" detecting and correcting routines one needs to supply. In the case of content errors and variants, it is rather the level and types of failure which the system's designers and users are willing to accept. If the system doesn't help, nothing can. In the following section we will examine the nature of the possible retrieval aids.

1.3.3 Compensating for Formal Errors, Content Errors, and Variants

A common characteristic of formal errors, content errors, and variants is that the values to be matched are "close" to each other in some sense. The problem for the system in compensating for them lies in the fact that not only will the concept of closeness between values vary from field to field, but also within a field it may be differently defined for different types of mismatches. A query which has a formal error in it may be "close" in different ways from one which has a content error or variant, and even different types of content errors and variants may not necessarily be compensated for easily by the same device. Consider, for example, a name string, "Smythe, John Dolan," and the following possible query strings:

a. Smith, John D.

b. John Smthe

The first contains a content error (Smith for Smythe) and a variant (D for Dolan) but is close to "Smythe, John Dolan" in that it is a reasonable transcription of the same or phonetic string in English. The variant use of only the middle initial is acceptable under many descriptive rules. The second query contains a keying error and a violation of the order rules for entering names, both formal errors under our definition. This query is also "close" because, on the one hand, a letter was "just" omitted and on the other, the inversion was not made. The same algorithm which compensates for "Smythe" and "Smith" or "Dolan" and the use of the initial might also by chance match Smythe and Smthe, but the inversion would surely have to be corrected either externally or, if internally, by a different algorithm.

Formal errors, then, may be corrected inside or outside the system with the help of the rules. On the other hand, content errors often behave like variants, particularly for fields such as title, but it is probably hard to generalize about them. More research here would not be amiss. Variants may be corrected only by providing a means of expanding the search to "close" values. Therefore, in solving "the search problem," one important field of interest must be definitions or rules for similarity within value sets or similarity among values of the same value set. As we have indicated, these may vary widely for different fields and may be complicated to construct. Most bibliographic systems already have some at least rudimentary tools of this type. Common examples are subject indexing thesauri, authority lists for names, and, in some experimental systems, their formally constructed semi-equivalents.

1.4 Similarity Between Value Sets

1.4.1 Definition

In section 1.2.2.4 we defined the value set of a file of object descriptions as the set of value types associated with each attribute of the class of objects in the file, such as the set of all the different authors' names in a file. Closeness between pairs of values within a single value set, such as index tags in a file of bibliographic descriptions or similarity within value sets, was seen to be an essential concept behind different types of error compensating devices in file search. We will now extend this concept of closeness to similarity between value sets.

In the discussion of value sets in section 1.2, it was suggested that values within value sets might be close in a number of different ways. Two written names, for instance, might be similar either because they were "correct" English transcriptions of the same spoken string or because there was "only" one letter at variance because of a keying error. As the basis for our extrapolation, although others may also be useful, we will use only one sort of "closeness," statistical association. The measures of this class, which are derived from data about the co-occurrence of pairs of values of an attribute in object descriptions, are now used in experimental bibliographic systems for such purposes as elaborating or expanding searches by subject tag when the original search has failed to satisfy the user. Many such measures have been investigated in a limited way (5,6,8,9) but at the present time much research remains to be done on their properties and the conditions which may work to the advantage of one rather than another. The entries in the two-by-two contingency tables (see Fig. 2) which provide the data for these measures contain for a pair of values the number of records in which both values occur, each value occurs separately, and neither occurs.

FIG. 2: TWO-BY-TWO CONTINEGENCY TABLE

	V_2	Not V_2
V_1	V_1 and V_2	V_1 and Not V_2
Not V_1	Not V_1 and V_2	Not V_1 and Not V_2

One possible interpretation of the intuitive idea behind such measurements is that two values are similar if they are associated with, or isolate highly overlapping sets of records in a file. An extension of such an idea to entire sets would suggest the following rough definition:

> Two value sets are similar if they are able, as established by computation or operational data, to isolate highly overlapping sets of records.

That is to say that if one thinks of each value set as the keys of an index, and if one knows the proper key or combination of keys, the two indexes are similar if one can isolate roughly the same group of records through either index. Measures of value set similarity are therefore potentially one component of index similarity which could be considered in making design decisions about system structure, as we shall see presently.

1.4.2 The Three Types of Similarity Between Value Sets

Having in mind this intuitive idea of similarity between value sets, let us examine the concept in more detail. Specifically, we distinguish the following three types of such similarity between value sets: (1) *distributional similarity*; (2) *functional similarity*; and (3) *elemental similarity*.

1.4.2.1 Distributional Similarity

Two values sets are *distributionally similar* when they have similar statistical distributions in the file. For the present we suggest determining distributional resemblance by examining the following parameters:

a. The number of records containing non-null values of each set;

b. The number of occurrences of values of each set in the file;

c. The variance and expected value of the number of records per value (discrimination) (e.g., the average number of documents by the same author); and

d. The expected value and variance of the number of values from the one value set per record, (e.g., the average number of index terms per document).

The expected value and variance of the number of values of an attribute per record (d) is a figure frequently of interest for researcher, but probably of less direct interest to users. The other three types of data, however, supply the necessary raw statistics for determining the relative size of the ranges of the two attributes and their discriminatory capabilities. We use such information intuitively in searching. For example, suppose we know the title of a book (when the title seems distinctive) and the subject heading assigned to it, and we want to know the call number. We might search by title rather than subject heading expecting to retrieve many fewer unwanted records by using the former.

1.4.2.2 Functional Similarity

Two value sets are *functionally similar* if, by virtue of the distribution patterns of their tokens in the file, one could isolate the same sets of records using either index. For example, suppose that a system designer wished to know whether it was worthwhile to provide an index by classification numbers to a set of document records in addition to an existing subject heading index. If he finds that, for most sets of records retrieved by searching through the subject index, the same set or a highly overlapping one could be isolated by retrieving the records associated with the union of the class numbers occurring in the first set, he might consider the second index to a large extent redundant. Let us label these sets as follows:

Set A = the records retrieved by a subject heading, i.e., through a value in the first value set or index.

Set B = the records retrieved by the records associated with the union of the class numbers co-occurring with the subject heading; i.e., the union of the values of the second attribute co-occurring with the value from the first attribute.

If A and B tend to be the same through many observations, then the designer might consider providing only a "translator" from class number to subject heading rather than a second index. The translator from subject heading to class number would presumably be cheaper to maintain than a second index if the composition of the two value sets and the associations between them were fairly stable. This stability does not seem an unreasonable hypothesis. The translator might range in accuracy and complexity from an index to all values co-occurring with the values of the first index to Boolean functions of values of the second set for more exact approximations. The cost of compiling and maintaining an elaborate translator, however, might eat up anything saved by not maintaining an index, but it seems worthy of consideration for some uses. Figures on the comparative use of the two value sets as search keys as well as the cost constraints of the system and the user would be factors influencing his decision.

An additional reason to find measures of functional similarity is to provide search aids. A searcher who has retrieved a Set A in response to his original query might wish to know whether it is "worthwhile" for him to expand his search by retrieving the records associated with the union of the class numbers which he had found in Set A, that is, Set B. Consistently large overlap between A and B would suggest that this extension would not likely be very productive of new references. For purposes of search expansion, "worthwhileness" would depend on the user's being able to expect several additional references in B. The most obvious statistic to aid him in his decision would be the number of records in B but not in (A AND B); that is, the size of the complement of the intersection of (A AND B) in B. The distribution of this statistic might be a reasonable basis for a measure of similarity.

Functional similarity is dependent on distributional similarity in important ways. For instance, so long as the value set of Set B is distributed with only one value per record, the union of those values is the only Boolean function which can be used to define Set B. Intersection and complementation of the values of that set cannot occur unless there is more than one value of that set per record. As soon as more than one value per record occurs, however, intersection and complementation of values of the second set can produce a closer match to A. More generally, let us call the degree to which any set of records may be specified by the value set of a field that field's degree of flexibility. For example, the records for two books with different authors cannot be specified as a set; that is, those and only those records, using the values of the author field alone unless it happens that each author only wrote one book. It is sufficient for a field to be fully flexible that within the field each record contains a string which is unique to that record. At worst, in this case, one can specify any possible group of records in the file simply by listing these unique strings. Record identification numbers are an example of a completely flexible field, whereas in bibliographic records the author field is not. Shallowly subject-indexed files, such as those created by the Library of Congress, do not have completely flexible subject fields, but as more and more index terms are added to a record, the number of records for which the index terms as a set constitute a unique string will approach the whole file. How quickly the field becomes fully flexible is, thus, a function of the distributional characteristics of the value set; i.e., its size, the number of values per record, and the number of records per value. The allowable indexing and retrieval grammars also play a part. The addition of full Boolean logic or links or roles, for example, should increase the number of unique strings for the same size vocabulary and the same indexing depth. The distributional characteristics of the field, then, have a great role in determining the ability of one field to function, in our limited sense, like another.

An additional facet of this ability must be high correlations between values of the different value sets, the less flexible the field being compared, the more two-way correlation is necessary for this field to duplicate the retrieval behavior of the first one. It may in fact be that a good measure of the similarity between values within a single value set. This process would have three steps.

a. Calculate the association values using some appropriate measure between all co-occurring pairs of values where one value is from the first set and the other from the one being compared to it.

b. Average or otherwise summarize for each value in the first set the association value it has with each value in the set being compared to it which co-occurs with it. In other words, for every A, average its associations with the elements of B.

c. Average these averages for all values in the first set; i.e.,

$$\frac{1}{I}\sum_{i=1}^{I}\frac{1}{J_i}\sum_{j=1}^{J_i}a_{ij}$$

where a_{ij} is the association value of the i^{th} value in the first set with the j^{th} co-occurring term in the second.

Another possible measure is the mean of the distribution of (B-(A AND B)), as suggested above. There are, then, at least two alternative approaches to developing measures.

It seems doubtful if the measures as such will be very useful as search expansion aids, as opposed to serving as criteria for system design decisions. It seems probable that too much information is lost in the summarization process; therefore, either the size of (B-(A AND B)) for a particular A or the average of the associations of the terms in B co-occurring with A, i.e.,

$$\sum_{j=1}^{J_i} a_{ij}$$

would be more informative indicators. We have used A here as a single term, but the same techniques could be extended to A's which were Boolean expressions.

The proposed types of measures make sense only so long as the set being compared is not completely flexible. In this latter case, we know it is possible to duplicate exactly any A using the second set; indeed, that is what the standard search-key-to-record-number or location index is. The question now becomes, how difficult is it to do? Although developing measures for this case seems an interesting problem, it has not been pursued any further to this time.

1.4.2.3 Elemental Similarity

Elemental similarity can occur when two or more fields of a record have value sets whose values have common "meaningful elements." Examples of "meaningful elements" are such strings as dates and keywords in subject heading and title fields. Functionally these are the elements which would serve as the basis of a common index to the two (or more) fields. Since there is no clear theory of what constitutes a meaningful element, the clarification of this rather fuzzy definition will not be attempted here. In some cases, even characters might be of interest to the user, but more casually one assumes a larger unit. We will call a set of such strings for one field its

Elemental Set (ES) and the intersection of two such sets for different value sets a Common Elemental Set (CES). In our present situation let us assume that the user's information can be translated into more than one value set, because the user can express his needs in terms of the CES. For example, the user may know a group of keywords associated with a topic of interest. If these keywords are in the CES, they may be translated by some matching algorithms into, for instance, either a set of subject headings or a set of titles. Let us call these two sets of documents thus retrieved C and C'. Here there are two levels in the search and, thus, two aspects in determining the similarity of the two value sets. The first is size of the CES relative to the two elemental sets, and the overlap of each of these individually with the user's set of elements. The second is the similarity over many observations of C and C'. Note that C and C' are selected, not because of the co-occurrence of values as is the case in A and B, but because of the occurrence of elements. Thus search expansion using functional similarity and search expansion using elemental similarity might frequently yield quite different results. (See Fig. 3 and Fig. 4.) The criteria for using one approach as opposed to another when both are possible as between title and subject fields, is one of the facets of the problem which needs to be investigated.

The basic data needed to measure elemental similarity with regard to the Elemental Sets is the same as that described in observing functional similarity; that is, the size of each of the elemental sets and the size of their intersection. The useful data concerning C and C' should be the same as that needed about A and B. The primary complicating factor in the use of elemental similarity is, of course, that fields must be indexed and that queries using the index must then be translated back to the value sets. The nature of this matching function clearly can influence the values selected and thus the record sets C and C'.

FIG. 3:

FUNCTIONAL SIMILARITY

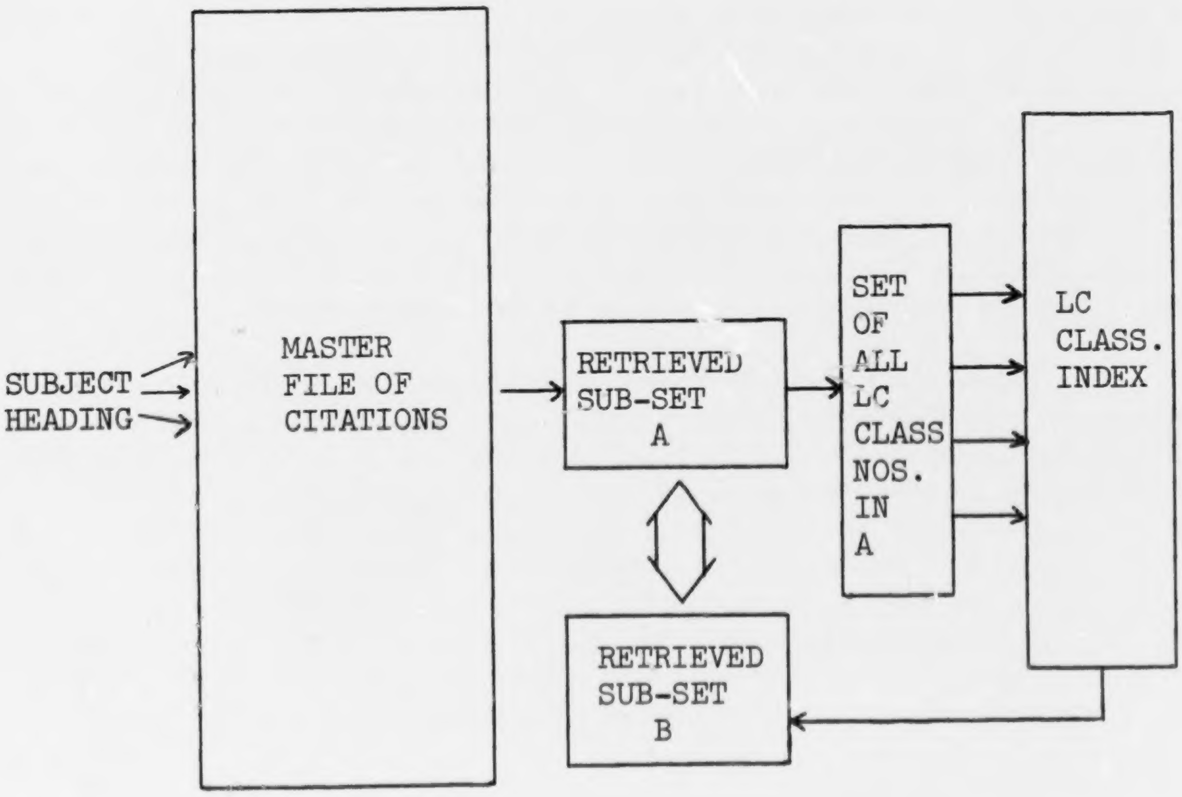
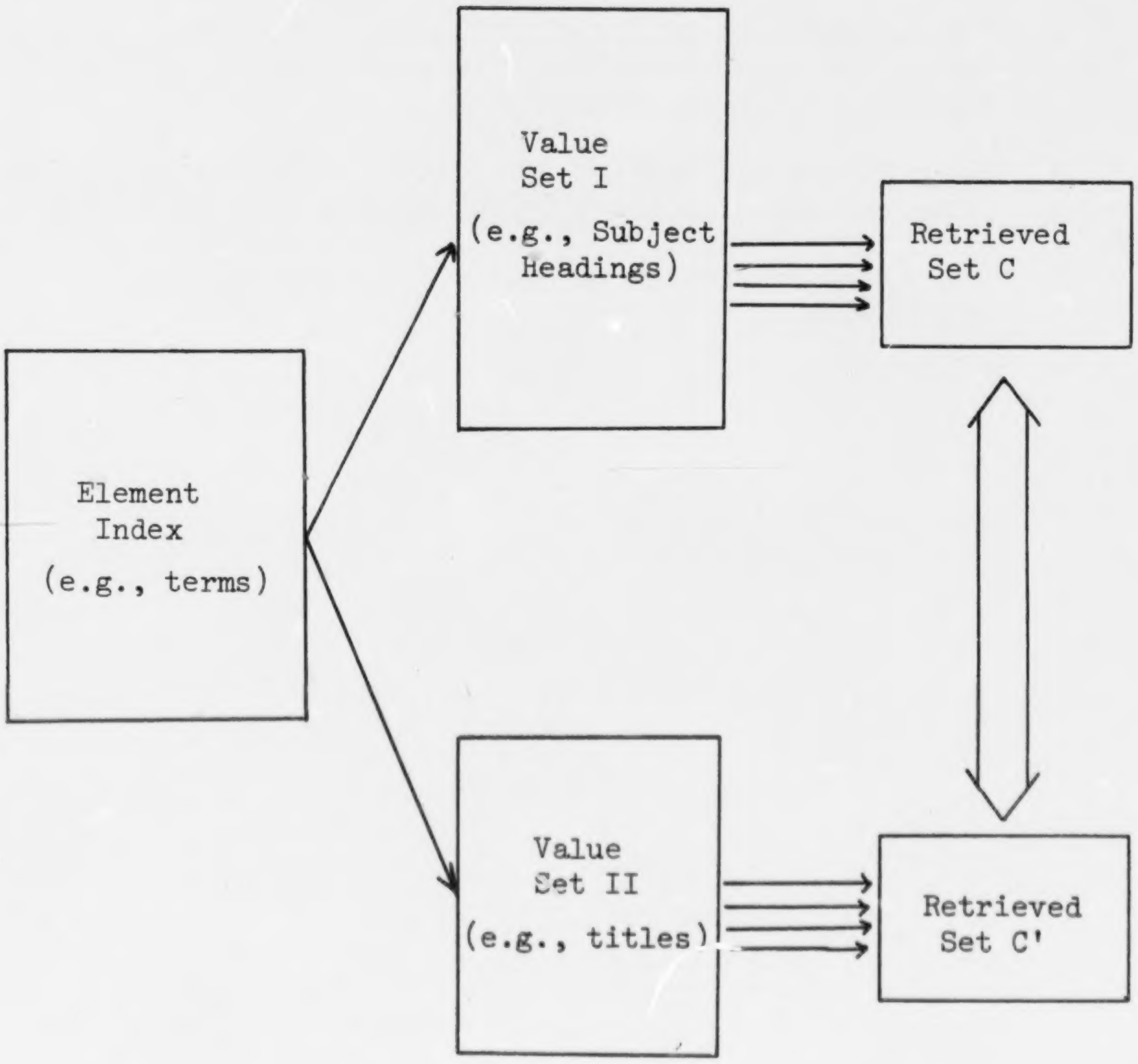

SUBJECT
HEADING
MASTER
FILE OF
CITATIONS
RETRIEVED
SUB-SET
A
SET
OF
ALL
LC
CLASS
NOS.
IN
A
LC
CLASS.
INDEX
RETRIEVED
SUB-SET
B

FIG. 4:
ELEMENTAL SIMILARITY


Value
Set I
(e.g., Subject
Headings)
Retrieved
Set C
Element
Index
(e.g., terms)
Value
Set II
(e.g., titles)
Retrieved
Set C'

1.5 Research

The following areas are suggested for research:

a. Measures for distributional, functional, and elemental similarity;

b. The relationship of distributional and functional similarity and the correlation between functional similarity and elemental similarity;

c. The relationship between functional similarity and inter-value set correlations between pairs or subsets of values;

d. The use of statistics on condition or set overlap as search aids.

REFERENCES

1. Minsky, Marvin, ed., Semantic Information Processing, Cambridge, Mass.: MIT Press, 1968.
2. Simmons, Robert F., "Natural Language Question-Answering Systems," CACM (13, January 1970), 15-30.
3. Travis, L.E., "Analytic Information Retrieval," In: Paul L. Garvin, (Ed.), Natural Language and the Computer, New York: McGraw-Hill, 1963, pp. 310-353.
4. Maron, M.E. and J.L. Kuhns, "On Relevance, Probabilistic Indexing, and Information Retrieval," JACM (7, July 1960), 216-244.
5. Stiles, H.E., "The Association Factor in Information Retrieval, JACM (8, April 1961), 271-279.
6. Kuhns, J.L., "The Continuum of Coefficients of Association," In: Statistical Association Methods for Mechanized Documentation, Washington, D.C.: National Bureau of Standards, (National Bureau of Standards Miscellaneous Publications 269), 1965, pp. 33-39.
7. Salton, Gerard, Automatic Information Organization and Retrieval, New York: McGraw-Hill, 1968.
8. Mignon, Edmond and Irene L. Travis, LABSEARCH: ILR Associative Search System Terminal Users' Manual, Berkeley: Institute of Library Research, University of California, 1971.

2. THE ASSIGNMENT OF INDEX TERMS

By Marcia Bates

2.1 Summary of Study

A 60,000 document file consisting of the first cataloging done for the library of the new University of California campus at Santa Cruz was analyzed by computer to gather data about the basic characteristics of the file, and to test two minor and one major hypothesis about the behavior of subject headings in the file.

The first minor hypothesis was that during the depression period catalogers gave more subject headings per document but did not add more than the usual number of new headings to the thesaurus. We found that more new subject headings were added to the thesaurus relative to the number of documents indexed during that period, but the conjectured causes were equivocal.

The second minor hypothesis stated that the mean date of publication to which a given subject heading is applied is indicative of the relative age of the subject field described by the heading (relative to other subject fields described by other headings). A random selection of headings and their mean dates was produced and a visual inspection gave one little sense of the progress of time.

The major hypothesis stated that documents with a high number of terms assigned represent new areas of activity where the vocabulary is not yet well developed (hence confusion on the part of indexers and a tendency to assign more headings). Two major tests were made on this hypothesis. In the first test the mean date of publication over all the applications of that term (or "heading") was computed, and this mean date was subtracted from the publication date of each individual application of that heading. This was done for all headings in the file. Then the mean number of terms assigned along with a given term was computed over all term applications (regardless of term) which had the same difference between mean date and date of publication of individual application. The second test did the same except the difference computed was that between the earliest date of application of a term and the date of individual application rather than the mean.

The first test appeared to confirm the hypothesis. However, there were several faults in the test and the second, less faulty test was then conducted. It produced negative results, i.e., yielded nothing to confirm the hypothesis. Finally, it appears that the critical test has not yet been performed. Therefore, some improvements on the current tests are discussed.

2.2 Description of Study

2.2.1 The Hypotheses

One would expect that when confusion exists over the naming of a subject field or sub-section of a subject field, this confusion would be reflected in the assigning of index terms to documents on that subject. One might further expect that this confusion would be demonstrated in that indexers would tend to give more terms to documents on such a subject. In other words, where there was no single generally agreed upon term, indexers would use several terms to be sure they had covered all the likely access terms to be used by index users.

While this problem might occur in any field or sub-section at any time, we would expect it to be especially acute in newly developing fields. So the hypothesis: Documents with a high number of terms assigned represent new areas of activity where the vocabulary is not yet well developed.

The hypothesis and the expected results it posits are, of course, independent of guesses made as to the cause of the results. The reason given above, that indexers feel they must assign several terms to cover all the likely access terms to be used, may not be the cause, or it may be one among several causes for the results. Here are two other possible explanations:

1. When working with a limited vocabulary, i.e., out of a thesaurus, there may not have been any apt term incorporated into the thesaurus yet to describe the new field. Among the terms in the thesaurus, there may be several each of which only partially describes the field. The indexer then may feel it necessary to use most or all of these to adequately describe a document's subject.

2. The first explanation, proposed above in presenting the hypothesis, presumed general terminology confusion in the field, with the consequence that the indexer must provide for all the possible access terms. However, even after opinion has largely consolidated around a particular term or set of terms to describe a field among the practitioners in a field, confusion may linger among non-specialist indexers. With some thesauri, such as the Library of Congress List of Subject Headings... there may be a relatively long lag before new terms are introduced to the list. Such terms will likely be new to the indexers and they may feel hesitant at first in dropping all at once the several terms formerly used in favor of the new term or set of terms, especially if they feel uncertain about how the new terms should be used. Their solution (conscious or unconscious) may be to retain some of the older terms, along with the new ones, in their indexing of the new field for a while.

John Tinker, writing in American Documentation, has some ideas related to the above. His views will be discussed in the next section.

In the process of developing tests for the above hypothesis, several other minor hypotheses were developed. There was time to carry out work on two of the three principal ones and they will be described here. The other will be discussed under future work in section 2.2.5.

1. A basic analysis of the data base revealed that the average number of subject headings assigned per document was significantly higher between 1930 and 1945 than either before or after. A question which

then arises is, is this solely because people had more time and fewer books then because of the depression, and so assigned more headings, or did they also add more than the usual number of new unique headings to the thesaurus during this period? Rather than speculating on the why's of this phenomenon now, let us simply state an hypothesis and see if it is confirmed: During the depression period catalogers gave more subject headings per document but did not add more than the usual number of new headings to the thesaurus.

2. A new book can be written about an old subject but an old book cannot have been written about a new subject. So the hypothesis here is that the mean date of publication of books indexed under a term is indicative of the relative age of the subject field. Note that it is relative age--because old fields can have new books written about them.

2.2.2 Background

A quite thorough literature search produced only three articles even reasonably related to the topic of this study. Two were purely statistical studies to determine basic characteristics of large files, and not made to test any hypotheses about the character of literature or indexing. The data these studies produced will be discussed in connection with the corresponding data from this study later on.

The third study, by John Tinker (4), relates to the hypothesis of this paper. (This is the first of a two-part series of articles. However, the second (3) branches into other areas and holds nothing of interest to us here.) His chief concern is with indexer inconsistency. To quote the abstract: "Indexers, in choosing or assigning all words strongly associated with concepts of a document, assert that the document means the word; therefore, consistency of indexing measures the precision with which meaning is understood by the indexers." (p. 96)

He defines highly precise use of a term as application of that term to any one document by all the indexers in a study. (Such a term may be applied elsewhere inconsistently.) He found in a test using 9 indexers, 100 descriptors, and 50 abstracts to be indexed, that 19 descriptors were used highly precisely according to the above definition. Analysis of these descriptors revealed this: "Of the 100 descriptors, 15 describe concepts that were unknown only a few years ago. Five of these new words, or 33% of them, were among the 19 most precise descriptors, while only 16% of the older words were used precisely. It would be interesting to know if new concepts are understood more precisely than older ones as suggested by these data." (pp. 99-100) This is contrary to one of the possible causes offered above for confirming results on the hypothesis of this paper (no. 2), namely, that meaning of new terms is poorly understood at first by indexers. Whatever the cause of results confirming the hypothesis, it seems unlikely (though, as ever, not impossible) that these results would be due to a more precise understanding of the new terms (than of old terms) on the part of indexers.

Strangely, Tinker says elsewhere: "The difference in usage of new words and the tendency to use new words less precisely than old words fail to appear in Part III..." (p. 100) (emphasis mine). This is a direct contradiction of what he says earlier and in the abstract to the paper. As the "more precisely" statement appears in two places, it is probably the intended one.

One more, philosophical, comment with regard to Tinker's study: He states in various ways throughout his article views to the effect that low precision in use of words is due to lack of full understanding of the meaning of a term on the part of individual indexers (see particularly p. 101). It seems likely, however, that general confusion or lack of agreement in a field can exist, that no one can be said to know the one correct definition of a term (because there is no single agreed upon definition yet). For example, Tinker states, "If a given term is applied to a specific abstract by a large number of indexers, it is fair to say that those who do not apply the term do not fully understand its meaning." (p. 101)

But it may be that those who did apply it were just as unsure as those who did not. And maybe, in the matter of new terms, out of their insecurity, they applied more than the usual number of older terms along with them (cf. the third cause again). Indeed, Tinker notes elsewhere: "Twelve of the 100 descriptors on the list were used in 34% of the descriptor-abstract pairs. Of these often-used descriptors, only 1 was new. The other 11 descriptors, or 13%* of the total, were older words. This suggests that descriptors for older concepts tend to be used more frequently." (p. 100). They would indeed if they were used along with new terms to describe new subjects as well as old subjects.

2.2.3 The Data Base

The data base used was the cataloging data for the basic collection of the newly-founded Santa Cruz campus of the University of California. This data had been converted to machine-readable form and recorded on magnetic tape. It was received by the Institute about a year before the beginning of this study. The original base consisted of roughly 80,000 main entries, composed of two large sub-files. The first, comprising roughly 35,000 items, were entries for cataloging done under the New Campuses Program. This program is described in detail in the article by Voigt and Treyz (5). The purpose of this program was to select, collect, and catalog basic undergraduate collections for the three new campuses of the University, San Diego, Irvine, and Santa Cruz. The remaining entries were for cataloging done at Santa Cruz on material collected independently by that campus. The cutoff publication date for the NCP collection is 1964, whereas there are some Santa Cruz titles for as recent as 1967.

---

*These percentages do not appear to relate to anything in Tinker's paper.

For the purposes of this study, only the way the two projects handled subject headings is of interest. Both used current subject headings from the LC List of Subject Headings... If LC cards were bought, they were amended to current usage.* (5, p. 2207) Over two-thirds of the NCP books were in print and LC catalog cards were used for these. LC cataloging was also used for out-of-print books when available. In contrast, most of the cataloging done at Santa Cruz was original.

The only other major difference between the two portions of the file is that Mr. Black says he encouraged his catalogers to add subject headings of their own to the authority list, especially in science, where they felt the LC headings were not adequate. There is no way of knowing how many of the headings in the data base are of this sort.

The exact number of entries in these two portions (NCP and Santa Cruz) is not known, nor is there any easy way to tell which came from where. However, it is not necessary to know this anyway; the value of this information on the two portions is to give an idea of the general character of the file.

When the file was first received by the Institute, there were a number of problems in reading and processing the file and about 5% of the original entries were eliminated by formal methods on a computer (eliminating those with improper field lengths, etc.). This left a file of 74,732 entries. Of these, 14,571 were documents to which no subject headings had been assigned. This left a basic file of 60,161 documents, each of which had at least one subject heading assigned to it. Will Schieber, of the Institute, had created a new file off this basic one, which was composed only of each subject heading application, that is, each logical record in the file gave a subject heading, the number of the document to which it was assigned and some other data. A particular subject heading was repeated as many times as there were documents to which it was assigned and a particular document number appeared as many times as there were subject headings assigned to it. There were 103,038 such application records.

2.2.4 Tests on the Hypotheses

2.2.4.1 Basic Data on the File

Computer programs were run on the file on the IBM 360/40 and the CDC 6400 computers at the Computer Center of the University of California at Berkeley. Programming was done in the FØRTRAN Language.

The purpose of the first computer program was simply to gather some basic data about the characteristics of the file, particularly with regard to the distribution of the number of subject headings applied to documents.

---

*Information on the Santa Cruz cataloging practices was obtained in a telephone conversation with Mr. Donald Black, who was Head of Technical Processes at the UCSC Library from October, 1964 to February, 1967.

The total number of unique headings (types, as opposed to applications--tokens) in the file is 39,537 and the total number of applications is 102,614. This makes for a mean of 2.595 applications per term. (This count was made after the date duds had been removed from the file and a few other records had been eliminated in the process of writing new tapes off the original, so the total number of applications is smaller than previously.) The significant portion of the distribution of the number of unique headings applied X times and proportions of the total number of headings are given in Figure 5. The highest number of times any one term was applied was 194.

FIG. 5: DISTRIBUTION OF NUMBERS OF APPLICATIONS (TOKENS) OF UNIQUE HEADINGS (TYPES)

Number of applications	Number of unique headings	Proportion of all unique headings
1	26508	.6705
2	5336	.1350
3	2247	.0568
4	1277	.0322
5	827	.0209
6	588	.0149
7	430	.0109
8	367	.0093
9	252	.0064
10	209	.0053
11	169	.0043
12	139	.0035
13	115	.0029
14	99	.0025
15	70	.0018
16	73	.0018
17	78	.0020
18	61	.0015
19	61	.0015
20	55	.0014
21	45	.0011

(All succeeding values are below .001.)

In a similar study made at the Library of Congress (2) the mean number of applications per unique heading was found to be 17.5. The highest number of times any one heading was applied was 1,204 (p. 102). The study was made on a sample of 442 headings rather than on the whole file as in this case. They found that 83.5 per cent of the headings had fewer than 11 entries (or applications) (p. 101). The comparable figure in this study is 96.2 per cent.

From their write-up it would appear that there is a serious weakness in their sampling procedure. Dubester, the author, says that the same sampling procedure was used for subject headings as for authors. The selection method for authors was as follows: "In every twenty-second drawer of the catalog, the first author entry that was 2 inches from the front of the card tray was selected for the sample." (p. 100) The use of such a sampling procedure would have the result that those headings which had many entries under them would have a higher probability of being selected because they take up more space in the drawer. Yet each heading should have an equal chance of being selected. The result is that the sample is biased toward headings with more entries.

The same mistake was made on a similar catalog tray sample of headings done by the Institute a couple of years ago. When the mistake was discovered and the sample retaken properly, the mean rate of entries per heading went down markedly. If the sampling at LC was done in exactly the manner described, then the sample is biased to headings with many entries. Actually, the biasing may not be as bad in this case as it was in ours, because the sampling was done in the main catalog. There, all types of entries are mixed. If a non-subject added entry was hit first and one then advanced to the first subject heading one came upon, then this is no longer a function of the bulk of cards taken up by a heading. If a subject added entry is hit immediately, however, the above biasing weakness applies. Thus, Dubester's sample is probably biased toward large headings about half as badly as ours was. A change to the proper sampling procedure would bring the mean number of entries down and would raise the percent of headings with fewer than 11 entries--which in turn would bring the results closer in line with those gotten on this study. Considering the great difference in size and character of the two libraries, such closeness would actually be surprising.*

Means, mediana and standard deviations of date of publication, number of pages, and number of subject headings assigned per docuemnt were computed. (Detailed descriptions of how these were computed are given in Section 2.3.1) In the original keypunching the data had not been verified so a series of formal tests were made first on the date and page figures for each document to eliminate duds. For example, pages were flagged if alphabetic characters other than V (for Volume) appeared in the page columns.

---

*See Section 2.3.2 for mention of another cause of difference between Dubester's and this study.

Incidentally, as it would be difficult to make a guess at average number of pages per volume, documents whose pagination was in volumes were flagged separately and not included in the calculations. As it happened, among the various tests for page duds, the test for the presence of a V came second, so there may be a few of the "regular" duds which are also items with pagination in volumes. The results were as follows:

Date duds: 262

Page duds: 1127

Pagination in volumes: 2618 (out of 60,161 documents)

There were 33 overlaps, that is, 33 documents which had both date and page duds. It is interesting to note that the expected number of overlaps, if the two error functions were purely independent, would be about 5.

The statistical calculations on pages and dates are based on that section of the file which is error free for that aspect. In other words, a document with a dud date and a valid page number is not included in the date calculations but is included in the page calculations. No tests were made on the number of subject headings because this was computed simply by counting records for that document. (Remember, there is a record for each subject heading application, not just for each document.) However, as the mean subject headings were calculated in relation to date blocks (see below) the sample base for this figure is identical to that for the date figures, i.e., it includes no dud dates.

First, figures for the whole file:

Mean	Number of documents
Date: 1953.76	59895
No. of pages: 309.83	56412
No. of Subject Headings: 1.713	59895

Next, various calculations were made for dates and number of pages for each subject heading level. A document at subject heading level 2 is one which has had two subject headings assigned to it. Listed in Figure 2 are the number of documents at each subject heading level in the file before duds were weeded. Along with it, for purpose of comparison, are the corresponding figures from a study made by Avram et al. (1) at the Library of Congress on a sample of entries from cards issued between 1950 and 1964. (Note that this is cards issued, the dates of publication are not restricted to this period, but probably do fall almost wholly in this range.) Their sample was restricted to the "regular series." To quote the authors: "This includes both monograph and serial material but excludes special-format materials, materials in oriental languages, and cards for words not cataloged by the Library." (p. 181) This would make it fairly similar to the Santa Cruz file.

FIG. 6: DISTRIBUTION OF DOCUMENTS BY NUMBER OF SUBJECT HEADINGS ASSIGNED

No. per Document	Santa Cruz File		Library of Congress Sample	
	Frequency	Per cent	Frequency	Per cent
0	14,571	19.5	319	14.3
1	31,456	42.1	1,192	53.6
2	18,371	24.6	551	24.8
3	7,353	9.8	127	5.7
4	2,160	2.9	29	1.3
5	791	1.1	6	0.3
6	26	0.03	0	0.0
7	2	0.003	0	0.0
8	2	0.003	0	0.0
9	0	0.0	0	0.0
etc.	74,732	100.0	2,224	100.0

Figure 6 gives all the remaining basic calculations made on the file and requires explanation. Means, medians, and standard deviations were computed at the first 6 subject heading levels. As 7 and 8 had only two documents in each, they were not included. The number of subject headings was computed for each of four date blocks, that is, the number of subject headings applied to all documents whose date of publication fell within the date block's range became the basis of computation for that block. This is the converse process from the computation for mean date at each subject heading level.

Finally, normal distributions were assumed for each of these three sets of data and calculations were made to determine whether differences between means were significant.* For example, mean dates at subject

---

*Using this theorem: If $\bar{x}$ and $\bar{y}$ are normally and independently distributed, then $\bar{x} - \bar{y}$ is normally distributed with mean $\mu_{x-y} = \mu_x - \mu_y$ and standard deviation $\delta_{x-y} = \sqrt{\delta_x^2/n_x + \delta_y^2/n_y}$. The null hypothesis was then tested ($\mu_{\bar{x}-\bar{y}} = 0$) using the standard conversion so that a standard normal table could be consulted: $t = \dfrac{\bar{x} - \bar{y} - \mu_{\bar{x}-\bar{y}}}{\delta_{\bar{x}-\bar{y}}}$. Values of t greater than the table values meant the null hypothesis was disproven, i.e. the difference between means was significant.

FIG. 7: CHARACTERISTICS OF THE FILE

	No. of headings applied	No. of valid date documents	Mean date	Standard deviation (in years)	Median date	Significance* .95	.99
DATE	1	31,289	1954.39	± 18.35	1960		
	2	18,313	1954.00	± 17.31	1960	S	N
	3	7,325	1952.63	± 17.18	1959	S	S
	4	2,154	1949.78	± 18.93	1957	S	S
	5	788	1944.57	± 24.51	1951	S	S
	6	26	1947.54	± 15.29	1948	N	N

	No. of headings applied	No. of valid page documents	Mean no. of pages	Standard deviation (in pages)	Median no. of pages**	Significance* .95	.99
PAGE	1	29,475	307.40	± 200.33	260-279		
	2	17,301	309.54	± 193.33	260-279	N	N
	3	6,928	311.85	± 185.22	260-279	N	N
	4	1,994	329.06	± 185.31	300-319	S	S
	5	692	345.20	± 178.90	300-319	S	N
	6	22	315.14	± 109.79	280-299	N	N

	Date block	No. of valid date documents	Mean no. of headings	Standard deviation (in headings)	Median no. of headings	Significance* .95	.99
SUBJECT HEADING (by date block)	1 (thru 1929)	3,497	1.86	± 1.04	(Not computed because highest no. of headings so low)		
	2 (1930-1945)	5,299	2.05	± 1.13		S	S
	3 (1946-1959)	19,351	1.67	± 0.88		S	S
	4 (1960-1967)	31,748	1.67	± 0.86		N	N

*S = significant, N = not significant
**Calculated in 20-page blocks

heading levels 1 and 2, 2 and 3, 3 and 4, etc. are tested. As there is no *a priori* reason to assume that the mean date at any particular subject heading level will always tend in one direction, (for example, that higher subject heading level will have earlier dates), a two-tailed test is used. This goes for the mean number of subject headings as well. However, we would expect that if mean numbers of pages differ, they will differ in that documents with more pages will have more subject headings. Hence, a one-tailed test was used here. Results as to significance are listed on the table for the .95 and .99 levels.

2.2.4.2 The Minor Hypotheses

A very important point should first be noted and kept in mind throughout the discussion of the hypotheses. It is obvious that we are not studying the entire publishing and library field. What we see here is only a reflection of that wider field in a very small library collection with its own idiosyncratic nature. It is an undergraduate collection, overwhelmingly composed of recent imprints, and is a "basic" collection of important books, rather than a research collection.

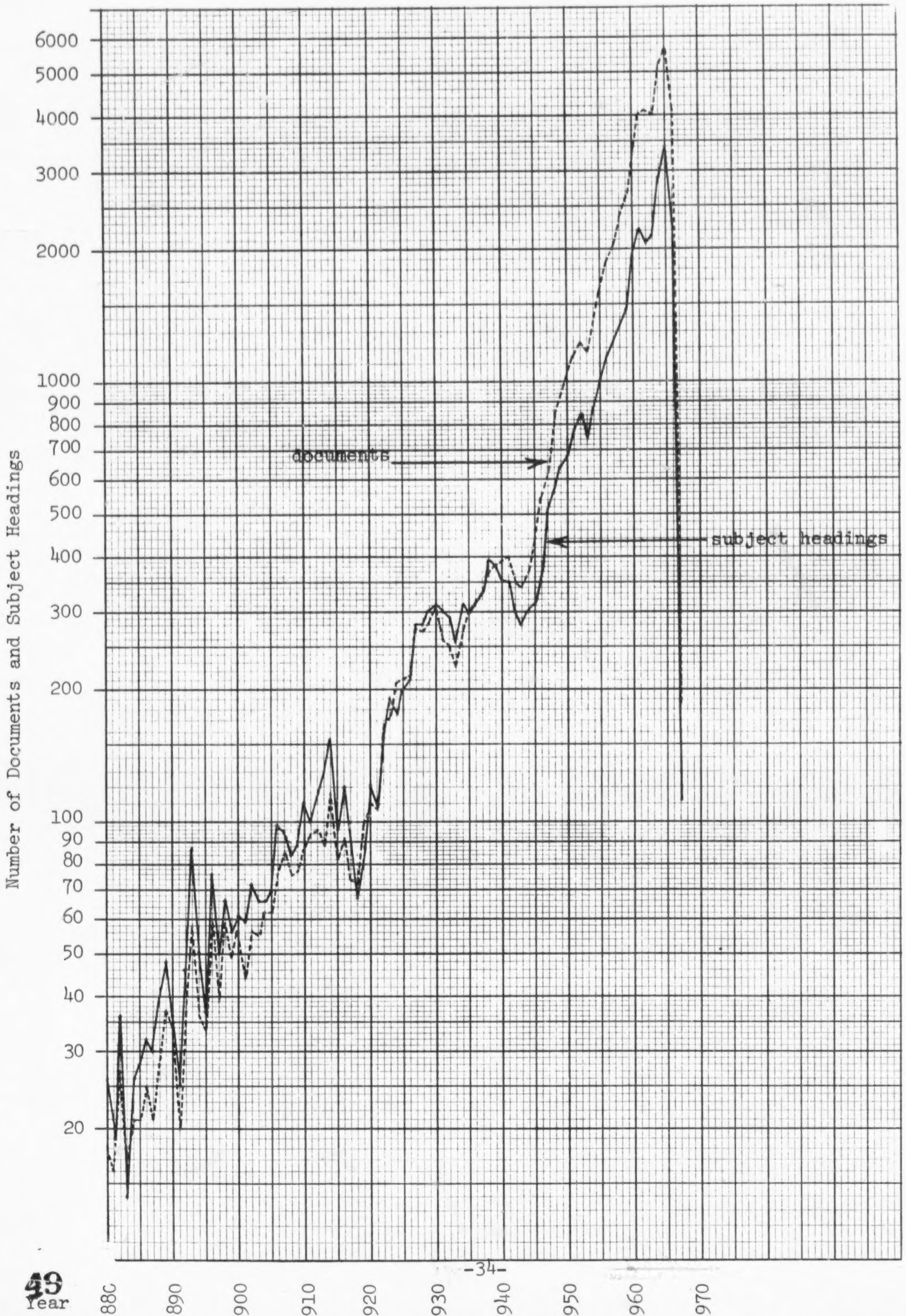
However, these books were not selected by the number of subject headings they would require, nor by the date, which are the two major bases upon which these tests were made. In other words, no conscious human bias affects the variables used as the basis for this study. Books were of course selected for recency, but they were not selected by date in the sense of trying to get a balance so that all the years are evenly represented or any such thing. Instead, selection was done on the basis of content; here is where the conscious human effort comes in.

Now let us examine the first minor hypothesis. The significant difference in the mean number of subject headings assigned per document between the various date blocks has been noted previously. The issue to be tested here is whether more than the usual number of new headings were added to the thesaurus during the period of the high rate of subject headings assigned per document.

Figure 8 graphs (on a logarithmic scale) the number of documents and number of new subject headings for the period 1880-1967. "Number of documents" is the number of documents whose date of publication is in the given year. "New subject headings" is the number of subject headings whose earliest date of application in the file is in the given year--in other words, the earliest date of publication over all the applications of a given heading. For purposes of the experiment these earliest dates of application are being assumed to be the date of addition of the heading to the thesaurus, which of course, they are not. In many instances the heading may first appear in the Santa Cruz file long after it has been added to the LC List. But here too, as with books being written about old and new subjects, a heading cannot be used before it is invented, so the results should be at least a blurred replica of the true profile.

It is to be expected that there will be relatively more new headings earlier than later, because of the particular nature of our

FIG. 8: NUMBER OF DOCUMENTS INDEXED AND SUBJECT HEADINGS INTRODUCED, 1880-1967



"new" headings. As these are the earliest date of application of a heading in the file, chances are that more recent documents will have subject headings whose earliest date of application is earlier than the date of the recent documents--so these subject headings will appear earlier on the subject heading curve. This can be seen on the graph. The document curve moves below the subject heading curve at first, moves up to pace it, and then moves above it.

The area of our prime interest is in the thirties, however. This section is blown up in Figure 9 (on ordinary graph paper). Here, mean number of subject headings assigned in each year is graphed as well, on a different scale. While the two curves had been neck and neck for a while, and one would expect the document curve soon to rise above the subject heading curve, instead it dips in the late twenties and thirties with a marked lowering between 1931 and 1934. So it would appear that more new subject headings were added during the depression relative to the number of books than in other periods.

Perhaps librarians had a lot of time on their hands and invented new headings. Or, perhaps because of the economic squeeze, fewer books were published, and those that were published were the best, the most original of an ordinary year's crop. This would mean that just as many subject headings were created in those years as would be expected, given the previous shape of the curve, but the squeeze lowered the book publishing rate. In other words, the cause of the flip in the curve could be either that number of subject headings rose, or the number of books published fell; the data do not tell us.

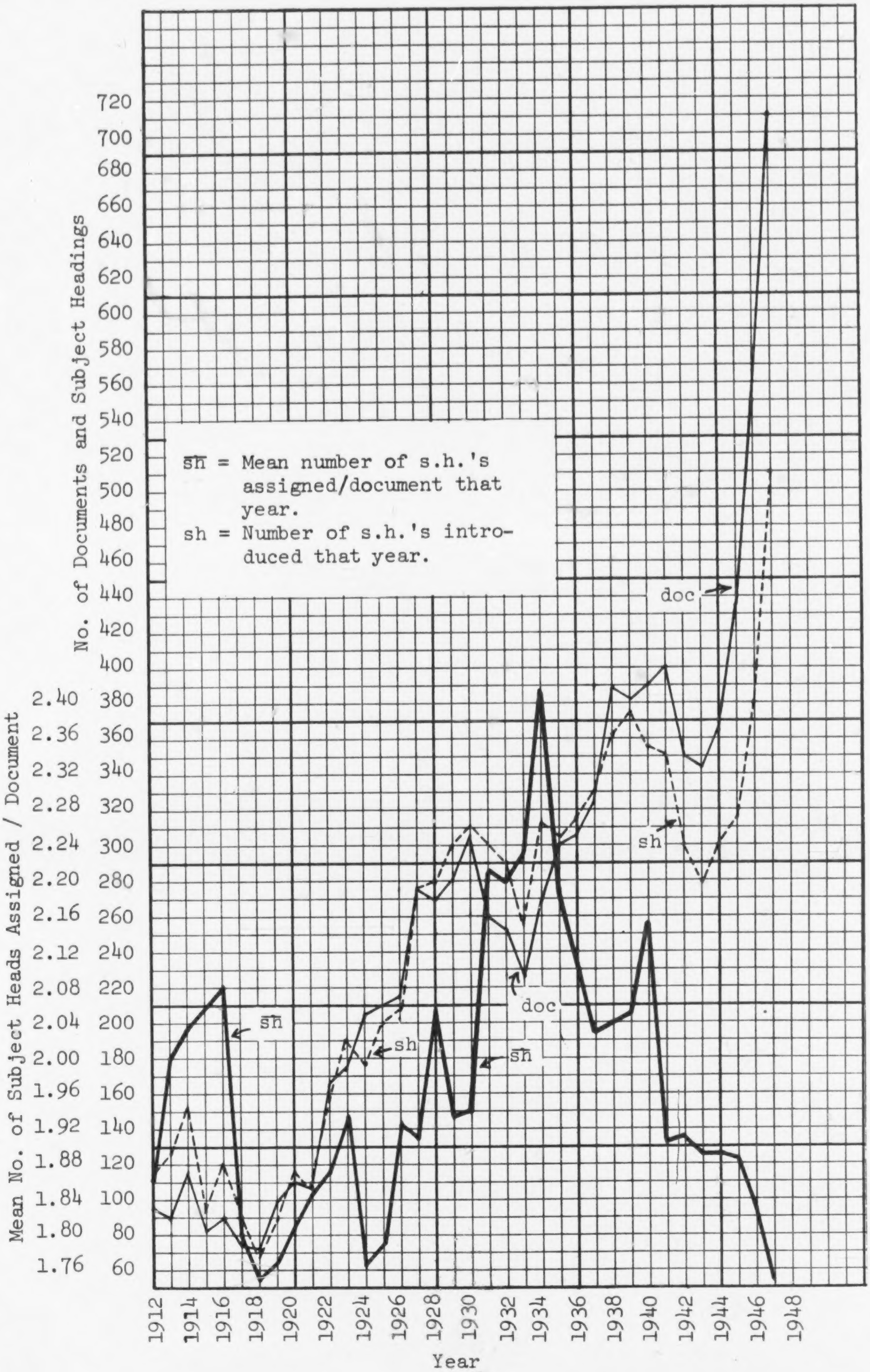
At first glance, it would appear that the opposite process occurred during World War II: more books but less time to bother about creating new subject headings. But referring back to Figure 7 again, we see that this is just the beginning of the general rise of the document curve above the subject heading curve. Whether it rises precociously in the forties, we cannot tell.

* * *

The second hypothesis, that the mean date of publication of books indexed under a subject heading is indicative of the relative age of the subject field, was to be tested simply by inspection of representative headings. A section of the file, ordered by mean date, was printed out, and then 100 headings were randomly selected from these. A detailed description of the method of selection is given in Section 2.3 and the list of 100 headings and their mean dates are given in Section 2.3.2.

The results indicate that a test of this sort is of little value on a group of subject headings of this nature. The hypothesis would be most sharply revealed with a collection that was largely scientific. The great majority of the 100 headings are geographical, personal, and historical. One gets very little sense of the progress of time in reading them over.

FIG. 9: BLOW-UP OF FIG. 8 WITH MEAN NUMBER OF SUBJECT HEADINGS ASSIGNED/DOC AND NUMBER OF DOCUMENTS AND SUBJECT HEADINGS (1912-1948)



2.2.4.3 The Major Hypothesis

The first approach taken on the hypothesis was to array the co-assignment and date of publication in a two-dimensional array. (Co-assignment, as used here, is the number of other headings assigned along with a given heading to a document. Looking at it in terms of documents, the co-assignment is always one less than the total number of headings assigned to a document.) With the array (x,y) all unique subject headings (types) whose mean date of publication fell in the date range x and whose mean co-assignment rate fell in the range y would be added to that array element. It was expected that the mean rate of co-assignment on headings with recent mean dates would be higher. This is a mistaken approach because the new heading phenomenon is going on constantly. A certain segment of the heading population in 1920 was new, just as a segment of the 1967 heading population is new. So over all the mean dates of subject headings the mean co-assignment should be roughly the same.

On the suggestion of Ralph Shoffner, a more sensitive approach was taken. The idea was to take the difference between the mean date of publication of a heading over all the applications of that heading, and the date of publication of the individual subject heading application. For example, the subject heading entries on the tape that was created to do this would look like this:

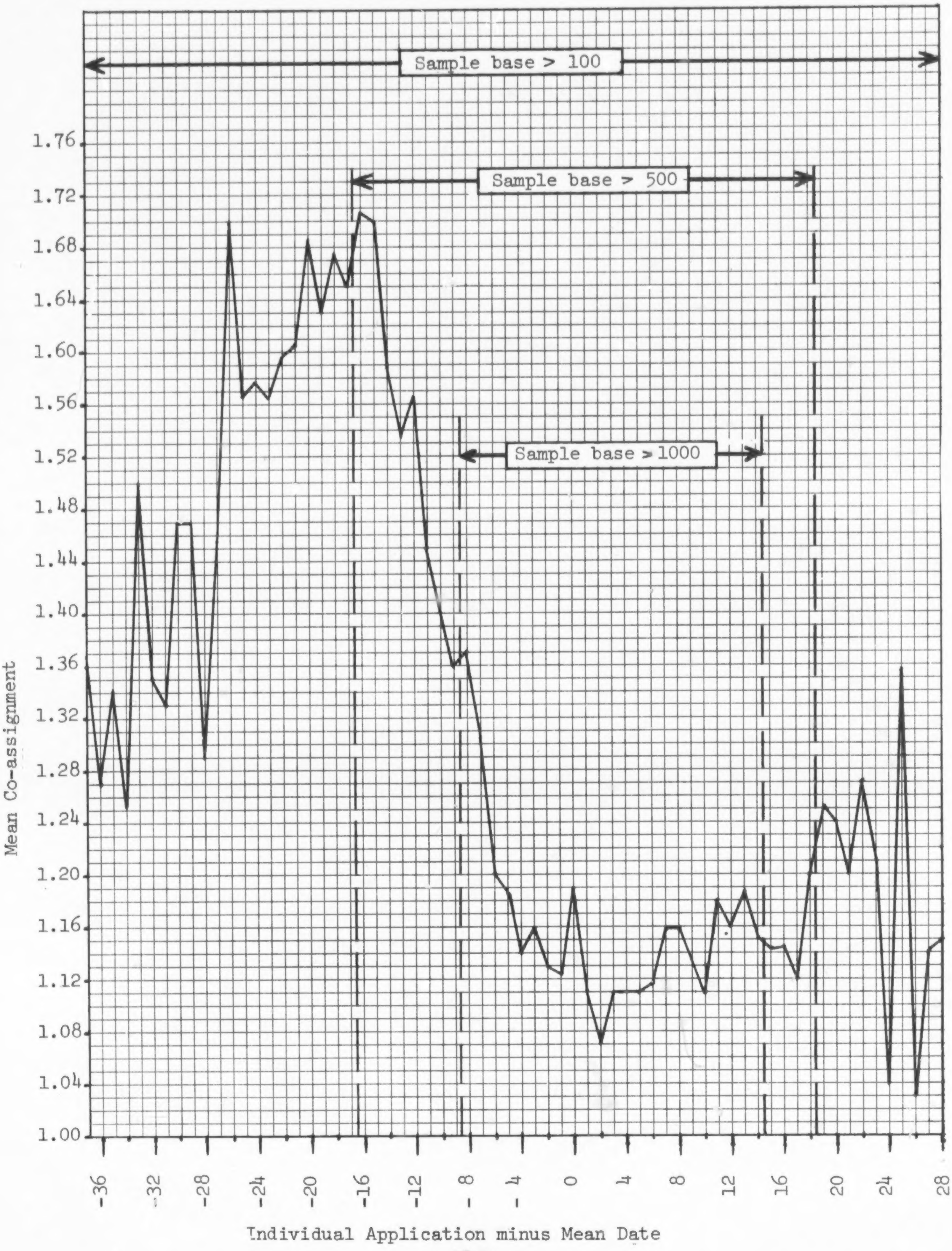
(subject heading)	(no. of co-as'd. terms)	(mean date)	(date of appl.)	(diff.)
Aesthetics	0	1960.0	1965	-5
Aesthetics	2	1960.0	1955	5
Aesthetics	1	1960.0	1960	0
Asps	1	1948.2	1950	-2

The tape was then sorted by difference, so that all entries with a given difference were together on the tape. Then the mean co-assignment rate over all those entries with a given difference was computed. The value of this method was that it blocked all headings applications by the distance from their mean date. This made it independent of date in the chronological sense; all those heading applications five years from their mean would be together, regardless of when in time the mean was. The results of this test are graphed in Figure 10.

A -20 means that the date of the individual application is 20 years earlier than the mean. At first sight, this graph is quite impressive; however, on closer examination, Mr. Shoffner and I found several faults with this approach. (This apparently simple study was sneakier than it first appeared, especially as the hypothesis and the assumptions behind it were not as clearly formulated at first.)

First, it was noted earlier that the mean number of subject headings assigned to documents varied over the years, with a rate notably higher during the thirties and early forties than during the periods before or after. (See Section 2.3.2 for the mean rate year by year.) Through 1929 it was 1.86, 1930-45 it was 2.05, and 1946-67 it was 1.67. With a mean

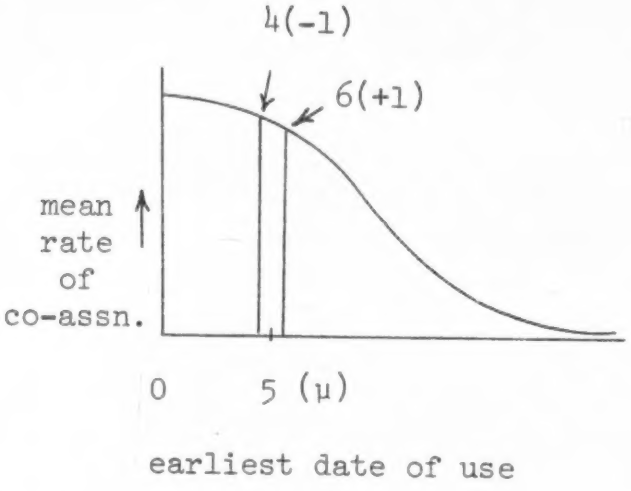
FIGURE 10: MEAN CO-ASSIGNMENT FOR DIFFERENCES BETWEEN MEAN DATE AND DATE OF INDIVIDUAL APPLICATION



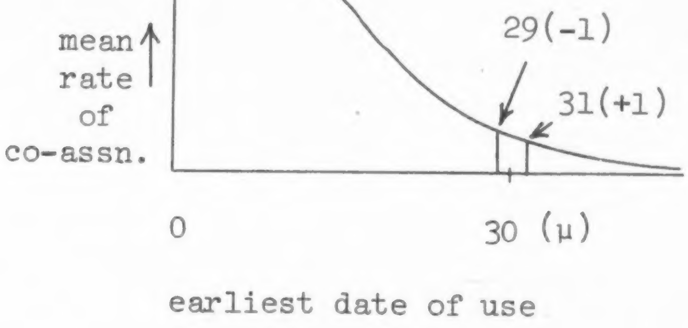
publication date of 1953.8 over all the documents in the file, it is evident that the vast majority of the collection is of recent date. So we can expect most subject headings to have a recent mean date as well. Thus the graph describes the behavior of recent books and their subject headings for the most part, simply because most of the collection is recent. So while the graph is technically independent of chronological date, in fact, most of the individual subject heading applications which are ten years earlier than their mean date (-10) are in the forties, ten years earlier than the date of the bulk of the file. So if we look at the range of highest mean co-assignment, -11 through -27, this would put us back right into the period where, as a general rule, the most subject headings were assigned to documents, hence the fall in the curve. We might then expect that the true curve, say, normalized by mean number of subject headings assigned per year, would give something closer to a zero-slope curve.

Secondly, there were two fundamental weaknesses with this logically. For one, by graphing each subject heading by years, we are using a cardinal, or interval, approach for an ordinal problem. In other words, in using this approach, we are assuming that one year (or ten years or whatever) has the same effect on every subject heading. Yet the rate of decay of various headings, and the resultant curves, may vary sharply from one to another.

The second logical fault may be even more serious. Here, the problem is in using the mean. Suppose on Subject Heading A, the mean is 5 years more recent than the earliest date of use of the heading, while for B the mean is 30 years from the earliest date of use of B. Suppose we then look at all subject heading applications which are one year earlier and one year later than the mean of Subject Headings A and B, i.e., -1 and +1.



Subject heading A



Subject heading B

Whatever the nature of the curves for the two subject headings (rate of decay), chances are that the -1 and +1 mean rates of co-assignment are going to hit at very different points on the curves. Yet this approach, by averaging together all co-assignments for Subject Headings A and B will blend these together and produce a rather blurred curve, that is, one would expect a curve such as in Figure 10 to have less of a fall in it than there really is.

One way of resolving this is to take the difference between date of publication of the individual subject heading and the earliest date that subject heading appears in the file. We then have the time from first use so that all values with a difference of 20 are 20 years more recent from their first appearance, which would then put all these values on roughly the same point in the decay curve of each subject heading. This eliminates the second problem, using the mean, but still does not solve the interval problem. Because of different decay rates, +20 may still be a different point on an individual heading's curve. There is also the problem that first use in the Santa Cruz file does not mean first use altogether--we may not get the heading until years after the initial confusion has died down at the Library of Congress. However, this problem is with us whatever approach is taken as long as we are using the Santa Cruz file and not the Library of Congress file! And anyway, we would expect a trend to be evident here too, because you cannot assign a heading which has not been invented yet.

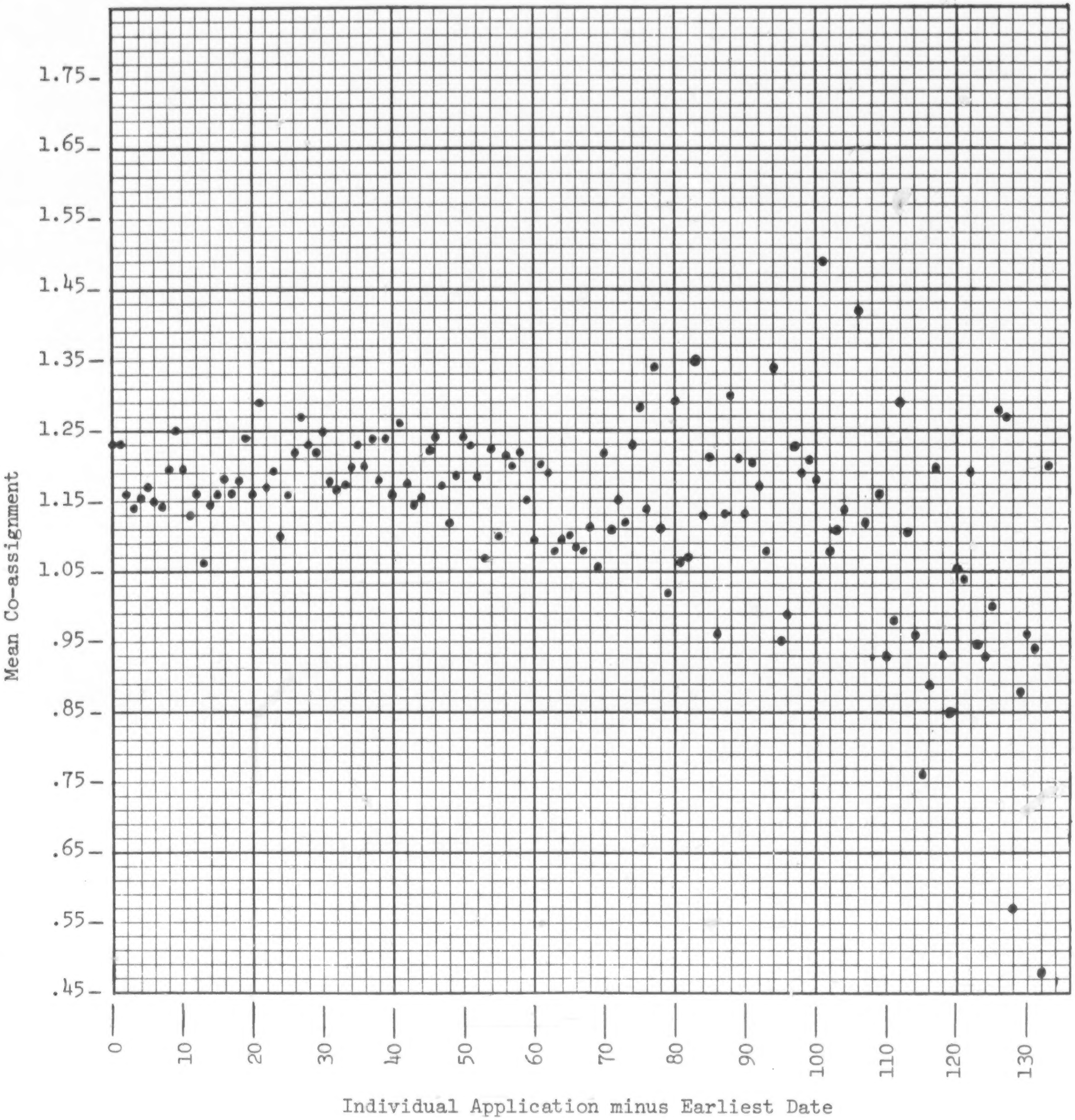
There is a way of getting around this interval problem too. However, there was not enough time to perform that test--so it will be discussed in the next Section, 2.2.5. Because of the very similar processing involved, there was enough time to do the intermediate experiment, (which eliminates the second problem but not the first), the one using the earliest date instead of the mean. The results of this test are graphed in Figure 11. A +20 means that the date of application of the individual heading is 20 years more recent than the earliest date of application of the heading. A glance indicates that the hypothesis is not at all confirmed in this test. The averages bounce around so that the lines of the graph were not evenly drawn in. The rates disperse more at the right hand of the graph simply because the sample bases were becoming very small there (from about 40,000 on the left to 20 on the right.)*

The interval problem still remains, but unless decay curves are radically different from heading to heading, it seems that there is no reason to believe the hypothesis to be correct. The impressive fall in the curve on the first approach is probably wholly due to the factors conjectured earlier.

---

*The number of heading applications at 0 were very high on both the mean and earliest date studies. This is because the great majority of the headings are applied only once, so the mean and earliest date are equal to the date of individual subject heading application. Plus 1 on the mean study drops to about 4400 and on the earliest date study to 2000.

FIG. 11: MEAN CO-ASSIGNMENT FOR DIFFERENCES BETWEEN EARLIEST DATE AND DATE OF INDIVIDUAL APPLICATION (SAMPLE BASE ⪆ 20 ONLY)



Some comments on the nature of the file and its relationaship to the tests made are in order here. The Santa Cruz file was used because it constituted a large mass of data in easily manipulable form. Yet it, by its nature, was not likely to be very revealing of the trends expected. First, it was indexed under the Library of Congress system and the number of terms assigned per document under that system is generally very low, thus allowing little leeway on the part of catalogers, and requiring us to descriminate very small differences on results. Also, as it is an implicitly hierarchical system, the solution resorted to when it appeared that many headings would be needed may have been to assign a single heading higher in the hierarchy. This is a common practice and a part of library students' instruction. A non-hierarchical system may evince the pattern hypothesized.

There are two other characteristics of this particular file which may make it unsuited for our purposes. First, it was mentioned earlier that subject headings previously assigned were revised to current usage. This could have had the effect of cancelling the confusion-resulting-in-more-headings for all periods except the present one. Secondly, the effect of Mr. Black's encouragement of his catalogers to use more headings where needed is unpredictable. This, in effect, results in a system which is neither purely Library of Congress nor a non-hierachical system.

Still and all, one persists in the feeling that were the trends there, they would have shown up. The approach suggested in the next section for future work should still be carried out, as it would eliminate both the fundamental weaknesses mentioned earlier and therefore provide more conclusive results. But aside from this it appears to this writer that the next best approach is to shift to bodies of literature which are indexed under non-hierarchical systems. The effects of such indexing systems are so different that they may well provide very different results.

2.2.5 What Next?

The following approach, mentioned earlier, was suggested by Ralph Shoffner to get around the interval problem. Order all the applications of each term by date of publication associated with that application, from earlier to later dates. Then go down the list for each subject heading and compare each pair of applications, 1 and 2, 2 and 3, etc. Look at the number of co-assigned terms in each pair. If the number of co-assigned terms on the first item of the pair is greater than the number for the second item, let us call that a yes; if less, a no. If the hypothesis is correct, then the percentage of all such pairs which are yes's should be significantly higher than the percentage of no's, i.e., higher co-assignment as a rule early in the life of the subject heading.

Another much more laborious approach would be to examine successive LC subject heading lists and supplements for new headings and use the date of the LC list as the earliest date. This would give us the true date of birth of the subject heading, but here the interval problem is back again. The results would probably not be much more interesting than the earliest-date-in-the-file approach already used.

It was mentioned earlier that several minor hypotheses were developed in the course of the work. Two have already been tested and discussed. The only other important one, which there was no time to test, is the following one. Where no new subject headings had been introduced to the thesaurus yet for a newly developing field, it is to be expected that indexers will use several older terms, each only partially descriptive of the field, to cover the subject adequately. Taking all the terms used in the file in pairs, we would expect that some pairs would be co-assigned (a different definition here, namely, both assigned to the same document) very frequently to describe newly developing fields for which single apt terms had not yet been introduced to the thesaurus. (There is nothing magical about pairs; threesomes may frequently be co-assigned as well. However, even examining all the pair co-assignments in the file becomes quite a bulky job.) The number of times pairs are co-assigned can be counted and pairs with high co-assignment printed out and compared to pairs randomly selected. Given the current rapidity of scientific development, most of the high-co-assignment pairs, representing new fields, should be in science.

2.3 Specific Tests and Data

2.3.1 Precise Description of Tests Made

The following is a precise description of the various calculations and tests made during the study. The descriptions are broken down in the same way as in Section 2. As the particular variables used in each test vary considerably, i.e., the same variables do not appear repeatedly in test after test, no attempt has been made to ensure that variables, when they *are* used again, have the same letter name.

2.3.1.1 Basic Data on the File

Distribution of Number of Applications over the File.

h = a unique heading (type).

a = an application of a unique heading (token).

t = total number of applications a of a unique heading h.

tt = total number of unique headings h which are applied t times.

th = total number of unique headings h.

Find the above totals over the whole file. Produce frequency distribution:

t	tt	tt/th (percentage)
.	.	.
.	.	.
.	.	.

Distribution of Documents at Each Subject Heading Level.

d_i = a document with i subject headings assigned to it.

Find the total number of documents with i subject headings assigned for each value of i, i = 1, n, and array the totals.

Means, Medians, and Standard Deviations of Dates, Pages and Subject Headings.

d_i = date of publication of a (valid-date*) document i.

p_i = number of pages of a (valid-page*) document i.

n = number of documents with valid dates.

m = number of documents with valid pages.

$\bar{d}$ = mean date of publication over whole file.

$\bar{p}$ = mean number of pages over the whole file.

Find:

$$\bar{d} = \frac{\sum_{i=1}^{n} d_i}{n} \qquad \bar{p} = \frac{\sum_{i=1}^{m} p_i}{m}$$

j = number of headings assigned to a document.

d_{ji} = date of publication of a document i with j headings assigned to it.

n_j = total number of documents with j headings assigned (valid-date).

$\bar{d}_j$ = mean date of publication of documents with j headings.

p_{ji} = number of pages of a document i with j headings assigned to it.

m_j = total number of documents with j subject headings assigned (valid-page).

$\bar{p}_j$ = mean number of pages of documents with j headings.

b = date block, b = 1,...,4; = thru 1929, 2 = 1930-45, 3 = 1946-59, 4 = 1960-67.

j_{bi} = number of headings assigned to a document i in date block b.

n_b = total number of documents in date block b.

$\bar{j}_b$ = mean number of headings of documents in date block b.

---

*Invalid dates and pages to be eliminated before calculation.

Find:

$$d_j = \sum_{i=1}^{n_j} d_{ji}, \text{ for values of } j,\ j = 1,\ldots,6.^*$$

$$\bar{p}_j = \sum_{i=1}^{m_j} p_{ji}, \text{ for values of } j,\ j = 1,\ldots,6.^*$$

$$j_b = \sum_{i=1}^{n_b} j_{bi}, \text{ for values of } b,\ b = 1,\ldots,4.$$

Also find: median date at each heading level j, and median number of pages at each heading level j.

s_{d_j} = standard deviation of date of publication at heading level j.

s_{p_j} = standard deviation of number of pages at heading level j.

s_{j_b} = standard deviation of number of headings in date block b.

Find:

$$s_{d_j} = \sqrt{\frac{\sum_{i=1}^{n_j} (d_{ji})^2}{n_j} - (\bar{d}_j)^2}, \text{ for values of } j,\ j = 1,\ldots,6.$$

$$s_{p_j} = \sqrt{\frac{\sum_{i=1}^{m_j} (p_{ji})^2}{m_j} - (\bar{p}_j)^2}, \text{ for values of } j,\ j = 1,\ldots,6.$$

$$s_{j_b} = \sqrt{\frac{\sum_{i=1}^{n_b} (j_{bi})^2}{n_b} - (\bar{j}_b)^2}, \text{ for values of } b,\ b = 1,\ldots,4.$$

---

*There were only two documents each for the values j = 7 and j = 8, so means were considered useless here.

2.3.1.2 First Minor Hypothesis: Rate of Addition of New Terms

Find earliest date of publication associated with each unique subject heading in the file.

h = a unique heading.

y = a year of publication.

th_y = number of unique headings h first appearing during given year y, i.e., that year is earliest date of publication associated with that heading in the file.

d = a document.

td_y = number of documents d whose date of publication falls in a given year y.

Graph frequency distributions of the above on the same graph, the number of unique headings and the number of documents on the y axis, the date on the x axis.

2.3.1.3 Second Minor Hypothesis: Relative Ages of Subject Fields

h = a unique subject heading.

dh_i = date of publication of a document i to which h has been assigned.

nh = total number of documents to which h has been assigned.

(m = total number of unique headings in the file.)

n = total number of heading applications in the file (not total number of documents), i.e.,

$$n = \sum_{i=1}^{m} nh_i$$

Find average date of publication of documents indexed under h, for each h.

$$\overline{dh} = \frac{\sum_{i=1}^{nh} \overline{dh_i}}{nh}$$

Then rank $\overline{dh}$ values:

$$\overline{dh_1} \leq \overline{dh_2} \leq \ldots \leq \overline{dh_m}.$$

Selection of subject headings for examination:

As the entire file is extremely lengthy, only portions of the subject headings whose mean date was a given year were printed out. If there were 20 or less in a given year, all headings were output. If there were more than 20, only the first 20 were output. Exactly how random this procedure is, is not known. The tape used, which was sorted by mean date, had been created off another tape which had been sorted alphabetically by subject heading. The sort routine was a packaged program and how the sorting was done, and what headings would tend to be among the first twenty is not known. Perhaps headings near the beginning of the alphabet tend to be more frequent. However, for purposes of our visual examination, any tendencies toward non-randomness which do exist are probably not significant. This output program produced approximately 2000 headings. Of these, the 100 headings reproduced in Appendix 2 were selected by a purely random process--in the usual way with a random numbers table.

2.3.1.4 The Major Hypothesis

First Approach--By Mean Date of Publication.

h = a unique heading.

dh_i = a date of publication i of a given unique heading h.

n = total number of applications of unique heading h.

Find mean date of publication for that heading over all applications of that heading.

$$\overline{dh} = \frac{\sum_{i=1}^{n} dh_i}{n}$$

Do this for all unique headings h in the file.

Find difference of mean date of heading and date of individual application

$$df = dh_i - \overline{dh}$$

for all applications in the file.

ch_{df_i} = number of headings co-assigned (other headings assigned to the same document) with a heading application i, which has a given difference df between the mean date of publication of the individual subject heading application.

m = total number of heading applications with a given difference value df.

Find mean

$$\overline{ch}_{df} = \frac{\sum_{i=1}^{m} ch_{df_i}}{m}$$

for each value of df.

Graph means on the y axis, values of df on the x axis.

Second Approach--By Earliest Date.

eh = earliest date of publication over all the applications of a unique heading h.

dh_i = date of publication of an application i of a given unique heading h.

m = total number of heading applications with a given difference value df.

Find earliest date of application eh for each unique heading in the file. Find difference of earliest date of heading and date of individual application

$$df = dh_i - eh$$

for all applications in the file.

Now, similarly to first approach, find mean number of headings co-assigned over all heading applications which have a given value of df.

$$\overline{ch}_{df} = \frac{\sum_{i=1}^{m} ch_{df_i}}{m}$$

Graph means on the y axis, values of df on the x axis.

2.3.2 Raw Data From Tests

On the following pages are listed those raw data from various tests which are not given elsewhere in the report. Due to the difficulties of xeroxing computer output paper, there is some overlap from page to page; please disregard all material above and below the horizontal lines drawn in.

2.3.2.1 Basic Data on the File

All pertinent data on this portion of the study have already been given.

2.3.2.2 First Minor Hypothesis: Rate of Addition of New Terms

The data are given in two separate tables. In the first, Table 1, the date, the mean number of subject headings assigned per document in that year, and the number of documents with that date of publication are listed. In the second, Table 2, the number of subject headings whose mean date of publication over all applications of the heading fall in the given year, and the number of headings whose earliest date falls in the given year are listed. The former, the mean, did not play a part in this test, but is left in here in case it is of interest.* Only the years graphed, 1880-1967 are listed.

2.3.2.3 Second Minor Hypothesis: Relative Ages of Subject Fields

Listed in Table 3, are the 100 headings selected more or less randomly from the whole file (see Section 2.3.1 for description of selection procedure) and ordered by mean date of publication.

Due to the lack of verification on the keypunching, and due to different practices with regard to punctuation between Santa Cruz and NCP, all blanks and punctuation marks were removed so that headings which were truly the same would sort together and tally correctly. The headings are reproduced in this compressed form.

There was not enough time to get an estimate of how many errors remained, i.e., how many "true" unique headings were considered as more than one heading by the program because errors in spelling caused divergence. It should be noted, however, that error in number-of-unique-heading counts will show up in figures which are more than the true number.

---

*In this calculation, by mistaken analogy with the usual practice of rounding numbers, mean dates were rounded in this manner: e.g., 1950 = 1949.5-1950.499...In the calculations for the major hypothesis this practice was dropped and e.g., 1950 = 1950.0-1950.99...,which is our usual way of thinking about dates.

2.3.2.4 Major Hypothesis

The difference, the mean number of co-assigned headings, and the number of applications which go into the calculation of the means are listed. Table 4 gives the data for the first approach, by mean date of publication and Table 5 gives the data for the second approach, by earliest date. Because of the sort program used, the positive and negative numbers are intermixed for the mean date data. Also, because of the way the subtraction was done, the signs on the original output were the opposite from the way graphed. So the signs have been changed to conform with the way graphed (but the data have not been altered!).

Table 1:
Rate of Addition of New Terms: Date and Mean Headings per Document

Date	Mean No. of Headings per Document in that Year	No. of Documents with that Date of Publication
1880	1.7222214	18
1881	1.8125000	16
1882	1.4814806	27
1883	1.5294113	17
1884	1.8571424	21
1885	1.6190472	21
1886	1.8799992	25
1887	1.8095236	21
1888	1.6071424	28
1889	2.0270262	37
1890	1.5882349	34
1891	1.6999998	20
1892	1.7631578	38
1893	1.8135586	59
1894	1.8378372	37
1985	1.4117641	34
1896	1.8983049	59
1897	1.7948713	39
1898	1.6130347	57
1899	1.7291660	48
1900	1.6607141	56
1901	1.8409090	44
1902	1.6964283	56
1903	1.7272720	55
1904	1.8548384	62
1905	1.5806446	62
1906	1.8289471	76
1907	1.7142849	84
1908	1.6842098	76
1909	1.7402592	77
1910	1.9647055	85
1911	1.6451607	93
1912	1.8631573	95
1913	2.0000000	89
1914	2.0341824	115
1915	1.8048773	82
1916	2.0888882	90
1917	1.7999992	75
1918	1.7500000	72
1919	1.7653055	98
1920	1.8108101	111
1921	1.8518515	108

Table 1:
Rate of Addition of New Terms: Date and Mean Headings per Document (cont.)

Date	Mean No. of Headings per Document in that Year	No. of Documents with that Date of Publication
1922	1.8773003	163
1923	1.9371424	175
1924	1.7621355	206
1925	1.7932692	208
1926	1.9252329	214
1927	1.9090900	275
1928	2.0551462	272
1929	1.9359426	281
1930	1.9405937	303
1931	2.2068958	261
1932	2.1865072	252
1933	2.2850876	228
1934	2.4157295	267
1935	2.1733332	300
1936	2.1111107	306
1937	2.0340557	323
1938	2.0408163	392
1939	2.0494785	384
1940	2.1560097	391
1941	1.9049997	400
1942	1.9085712	350
1943	1.8953485	344
1944	1.8964577	367
1945	1.8868351	433
1946	1.8330307	551
1947	1.7430553	720
1948	1.7072010	847
1949	1.6902456	933
1950	1.6379309	1044
1951	1.7144117	1131
1952	1.6797066	1227
1953	1.6355925	1180
1954	1.6150303	1304
1955	1.6604118	1599
1956	1.6534491	1841
1957	1.6688023	2029
1958	1.6435642	2323
1959	1.6903124	2622
1960	1.6631041	3369
1961	1.6634102	3993
1962	1.6197491	4071
1963	1.6787481	4028
1964	1.6432428	5180
1965	1.6781263	5701
1966	1.6821985	4056
1967	1.7189188	185

Table 2:
Rate of Addition of New Terms:
Mean Publication Date and Earliest Date of Appearance

Date	No. of Headings with that Mean Date	No. of Headings with that Earliest Date
1880	8	26
1881	7	19
1882	16	36
1883	9	14
1884	12	26
1885	10	28
1886	11	32
1887	8	30
1888	23	40
1889	22	48
1890	18	36
1891	12	25
1892	23	46
1893	38	87
1894	22	51
1895	15	37
1896	24	76
1897	27	50
1898	27	66
1899	23	57
1900	30	61
1901	22	59
1902	27	72
1903	32	66
1904	28	66
1905	37	69
1906	28	98
1907	51	95
1908	28	84
1909	41	88
1910	56	112
1911	54	100
1912	55	113
1913	59	128
1914	85	154
1915	51	95
1916	47	121
1917	44	88
1918	43	67
1919	42	88
1920	67	118
1921	60	108
1922	89	157
1923	114	192
1924	104	176

Table 2:
Rate of Addition of New Terms:
Mean Publication Date and Earliest Date of Appearance (cont.)

Date	No. of Headings with that Mean Date	No. of Headings with that Earliest Date
1925	103	198
1926	121	208
1927	139	274
1928	156	279
1929	166	300
1930	204	309
1931	180	302
1932	188	289
1933	190	256
1934	216	313
1935	224	303
1936	258	313
1937	279	329
1938	285	363
1939	290	384
1940	324	356
1941	303	351
1942	300	298
1943	308	279
1944	364	298
1945	352	316
1946	383	379
1947	541	511
1948	611	567
1949	675	640
1950	696	671
1951	805	767
1952	881	842
1953	911	753
1954	956	869
1955	1137	998
1956	1271	1156
1957	1430	1215
1958	1613	1333
1959	1747	1458
1960	2181	1920
1961	2411	2203
1962	2500	2079
1963	2589	2153
1964	3269	2908
1965	3591	3382
1966	2393	2246
1967	119	110

Table 3:
Relative Age of Subject Fields

Heading	Mean Date of Publication
GTBRITCOLONIESAFRICAWEST	1805.000
NATURALHISTORYOCEANICA	1811.000
LEEBOOD1784	1822.000
NEWSOUTHWALESAUSTCOMMERCE	1823.000
AFRICASOUTHDISCEXPLOR	1836.000
CHANSONDEGESTECOLLECTIONS	1859.000
DEBTSPUBLISUS	1865.000
COMMERCEPERIOD	1866.000
LOUISXVIIOFFRANCEFICTION	1867.000
ROMANCELANGUAGESETYMOLOGY	1870.000
CHARITIESUSCONGRESSES	1874.000
NEWGUINEADISCEXPLOR	1876.000
SCIENTIFICSOCIETIESBIBL	1879.000
POETRYOFPLACESYOSEMITEVALLEY	1880.000
SINDBADDTHEPHILOSOPHER	1882.000
MANUSCRIPTSENGLISHCATALOGS	1884.000
TASMANIADESCRTRAV	1885.667
RIVERSRIGHTOFNAVIGATIONOF	1886.000
FOLKLOREHUNGARIAN	1889.000
PUNICWARS	1889.000
SCOTCHINAMERICA	1889.000
SANLUISOBISPOCOCALIFBIOG	1891.000
MIRABEAUHONORECGABRIELRIQUETTICOMTEDE17491791	1891.000
PACIFICCOAST	1891.500
STMARTHOLOMEWSDAYMASSACREOF1572FICTION	1893.000
GRESSETJEANBAPTISTELOUIS17091777	1894.000
BERKSHIRECOMASSDESCRTRAV	1895.000
SANTACLARACOCALIF	1895.000
OREGONHISTFICTION	1900.000
MINNESOTADESCRTRAV	1901.000
SPECTRUMANALYSISBIBL	1902.000
ETCHING	1902.000
NETHERLS16481714FICTION	1902.000
LONDONROYALACADEMYOFARTS	1904.000
HAECKELERNSTHEINRICHPHILIPPAUGUST18341919DIE	1905.000
POLITICALSCIENCEDICTIONARIES	1906.000
WHITMANWALT181901892SERIESAMERICANMENOFLET	1906.000
TOLEDOSPAINDESCR	1907.000
LEARNEDINSTITUTIONSSOCIETIES	1908.000
BASKETMAKING	1914.000
MINIATUREPAINTINGHIST	1917.000
ESPERANTOGRAMMER	1917.000
GIOTTODIBONDONEI12661337	1917.000
WEBERKARLMARIAFRIEDRICHERNSTFREIHERRVON17861	1918.000
ARNOULDSOPHIEDRAMA	1919.000
WATERCOLORSBRITISHEXHIBITIONS	1919.000
FOLKLOREENGL	1919.000
EPICPOETRYFRENCH	1919.000

Table 3:
Relative Age of Subject Fields (cont.)

Heading	Mean Date of Publication
EUROPEANWAR19141918REGIMENTALHISTORIESUS103D	1920.000
FOLKLOREPHILIPPINEISLS	1921.000
MASSESTO1800VOCALSCORES	1923.000
CHILDRENASMUSICIANS	1925.000
ARCHIVESWESTINDIESBRITISH	1926.000
TRAVELMEDIEVAL	1926.000
FOUNDINGKINGSCOLLEGE	1929.000
PHILOSOPHYENGLISH13THCENTURY	1930.000
SPANISHAMERICARELATIONSGENERALWITHSPAIN	1931.000
SPANISHAMERICACOMM	1931.000
ROSSETTIGABRIELEPASQUALEGIUSEPPE17831854	1932.000
PASSIONMUSCITO1800VOCALSCORES	1932.000
FRENCHDRAMAMEDIEVALHISTCRIT	1933.000
GERMANLITERATUREMIDDLEHIGHGERMANBIOBIBL	1933.000
EUROPEANWAR19141918AFRICAGERMANSOUTHWEST	1933.000
NATIONALMUSICHISTCRIT	1934.000
FRANCERELATIONSGENERALWITHITALY	1936.000
FRONTIERPIONEERLIFENEWZEALOTAGO	1936.000
FRONTIERPIONEERLIFEILLINOISPIKECO	1936.000
NEWHEBRIDESHIST	1937.000
CALIFORNIAUNIVERSITYVIEWS	1937.000
BRAJLANGUAGEGRAMMAR	1938.000
BLACKHAWKWAR1832FICTION	1938.000
JOHNSONMARTINELMER18841937	1940.000
BEHAVIORISMPSYCHOLOGY	1943.000
THEWESTECONCONDIT	1943.000
LOWELLJAMESRUSSELL18191891	1944.000
PAINTERSCANADIAN	1945.000
HUAHINESOCLIFECUST	1946.000
USHISTWARWITHMEXICO18451848NAVALOPER	1947.000
UNITEDNATIONSPALESTINE	1947.000
REUTHERWALTERPHILIP1907	1949.000
AMERICANLITERATURECALIFORNIAHISTCRIT	1950.000
CRIMINALLAWCALIFORNIACASES	1951.000
ENGLRELATIONSGENERALWITHGENEVA	1952.000
EUGENICSHIST	1952.000
ARGENTINEFICTIONCOLLECTIONS	1953.000
FLAGELLANTSFLAGELLATION	1954.000
FERMIENRICO1901	1954.000
MINASVELHASBRAZIL	1956.000
EDUCATIONINDIA	1957.000
ENGLISHLITERATURETRANSLATIONFROMSPANISH	1957.000
ENGELBREKTENGELBREKTSSOND1436DRAMA	1959.000
ENGLISHLANGUAGEDICTIONARIESCZECH	1959.000
EGYPTIANLANGUAGEDICTIONARIES	1960.000
USRELATIONSGENERALWITHITALY	1961.000
PORTUALCOLONIESNATIVERACES	1963.000
POWERMECHANICS	1963.000
FOUNTAINALBERTJANNINGS18381895	1965.000
FERGUSSONROBERT17501774	1965.000
RESEARCHEUROPEEASTERNDIRECT	1967.000

Table 4:
Major Hypothesis Data: Mean Date of Publication

DIFFERENCE	MEAN CO-ASSN.	SAMPLE BASE
0	1.19130	31824
-1	1.12394	3905
-2	1.12912	3067
-3	1.15666	2464
-4	1.13944	1922
-5	1.18301	1683
-6	1.20145	1375
-7	1.30424	1180
-8	1.36639	1089
-9	1.36182	901
+1	1.10988	4441
+2	1.07346	4152
+3	1.11256	3767
+4	1.11108	3466
+5	1.11015	3132
+6	1.11578	2902
+7	1.15568	2640
+8	1.15684	2289
+9	1.13304	2067
-10	1.40069	871
-11	1.45317	726
-12	1.56250	656
-13	1.53795	606
-14	1.58667	525
-15	1.70054	551
-16	1.70588	544
-17	1.64795	463
-18	1.67483	449
-19	1.62763	427
-20	1.68462	390
-21	1.60382	366
-22	1.59249	346
-23	1.56949	295
-24	1.57382	298
-25	1.56800	250
-26	1.70089	224
-27	1.45411	207
-28	1.29167	192
-29	1.47340	188
-30	1.47273	165
-31	1.32749	171
-32	1.34848	132
-33	1.49565	115
-34	1.25424	118
-35	1.34210	114
-36	1.27103	107
-37	1.36274	102

Table 4:
Major Hypothesis Data: Mean Date of Publication (cont.)

DIFFERENCE	MEAN CO-ASSN.	SAMPLE BASE
-87	2.25000	4
-88	0.40000	5
-89	1.00000	4
-90	1.30000	10
-91	0.66667	6
-92	1.00000	1
-93	2.00000	3
-94	1.42857	7
-95	0.0	1
-96	1.50000	4
-97	2.25000	4
-98	0.66667	3
-99	1.80000	5
+10	1.10910	1769
+11	1.18155	1691
+12	1.15949	1486
+13	1.18402	1239
+14	1.15019	1072
+15	1.14379	918
+16	1.14578	782
+17	1.11755	621
+18	1.19669	544
+19	1.25000	416
+20	1.23876	356
+21	1.20323	310
+22	1.27197	239
+23	1.20895	201
+24	1.04348	161
+25	1.35404	161
+26	1.02655	113
+27	1.13861	101
+28	1.14912	114
+29	1.27586	87
+30	1.18518	81
+31	1.23077	78
+32	1.05660	53
+33	1.15517	58
+34	0.92308	39
+35	1.18421	38
+36	1.19444	36
+37	1.00000	30
+38	1.31818	22
+39	1.41379	29
+40	1.24000	25
+41	1.16667	18
+42	1.25000	20
+43	1.07143	14
+44	0.90476	21
+45	0.90909	11
+46	0.93333	15
+47	1.54545	11
+48	1.25000	16
+49	1.35714	14
+50	1.80000	10
+51	1.53846	13

Table 5:
Major Hypothesis Data: Earliest Date of Publication

DIFFERENCE	MEAN CO-ASSN.	SAMPLE BASE
0	1.23421	40988
1	1.23338	2031
2	1.15814	1897
3	1.14080	1875
4	1.15283	1819
5	1.16871	1713
6	1.15042	1549
7	1.14637	1462
8	1.19341	1427
9	1.25236	1379
10	1.18805	1372
11	1.13033	1289
12	1.16247	1231
13	1.05792	1174
14	1.14748	1173
15	1.16229	1029
16	1.18431	1020
17	1.16438	949
18	1.17534	941
19	1.23853	872
20	1.15643	863
21	1.28740	849
22	1.17150	828
23	1.18396	848
24	1.10270	925
25	1.16122	887
26	1.22161	907
27	1.27042	869
28	1.23114	822
29	1.22294	776
30	1.25196	766
31	1.17728	801
32	1.16809	821
33	1.17245	835
34	1.20187	748
35	1.23119	731
36	1.20341	762
37	1.24512	718
38	1.16227	721
39	1.24337	641
40	1.15916	622
41	1.26263	594
42	1.17647	561
43	1.14689	531

Table 5:
Major Hypothesis Data: Mean Date of Publication (Cont.)

DIFFERENCE	MEAN CO-ASSN.	SAMPLE BASE
44	1.15258	485
45	1.22336	488
46	1.24286	490
47	1.17474	475
48	1.12076	472
49	1.18610	446
50	1.24314	510
51	1.22993	461
52	1.18619	478
53	1.04728	423
54	1.22857	420
55	1.09850	467
56	1.21542	441
57	1.19775	445
58	1.21469	354
59	1.15168	356
60	1.09315	365
61	1.20414	338
62	1.19079	304
63	1.08191	293
64	1.09718	319
65	1.10239	293
66	1.08896	276
67	1.07807	269
68	1.11409	298
69	1.05556	270
70	1.21901	242
71	1.10849	212
72	1.15347	202
73	1.11940	201
74	1.23171	164
75	1.28492	179
76	1.14197	162
77	1.33793	145
78	1.11333	150
79	1.02069	145
80	1.29286	140
81	1.06803	147
82	1.07018	114
83	1.35000	120
84	1.13158	114
85	1.21428	112
86	0.96364	110
87	1.13483	89
88	1.29885	87

Table 5:
Major Hypothesis Data: Mean Date of Publication (cont.)

DIFFERENCE	MEAN CO-ASSN.	SAMPLE BASE
89	1.20690	87
90	1.13402	97
91	1.26596	94
92	1.17391	92
93	1.08036	112
94	1.34375	96
95	0.95455	88
96	0.99123	114
97	1.22785	79
98	1.19000	100
99	1.21428	98
100	1.17857	84
101	1.49123	114
102	1.08140	86
103	1.43678	87
104	1.11224	98
105	1.13483	89
106	1.35632	87
107	1.12500	64
108	1.41791	67
109	1.15873	63
110	0.93182	44
111	0.98333	60
112	1.29091	55
113	1.10417	48
114	0.96000	50
115	0.76190	42
116	0.88889	45
117	1.19512	41
118	0.92683	41
119	0.82759	29
120	1.05556	36
121	1.37143	35
122	1.21428	28
123	0.94444	18
124	0.93103	29
125	1.00000	25
126	1.28000	25
127	1.27273	22
128	0.57143	28
129	0.87879	33
130	0.96296	27
131	0.94444	18
132	0.47619	21
133	1.20000	20
134	1.11111	9

REFERENCES

1. Avram, Henriette D., et al., "Fields of Information on Library of Congress Catalog Cards: Analysis of a Random Sample, 1950-1964," The Library Quarterly, (XXXVIII, April 1967), 180-192.

2. Dubester, Henry J., "Stuides Related to Catalog Problems," In: Ruth F. Strout (Ed.), Library Catalogs: Changing Dimensions, Chicago: University of Chicago Press, 1964.

3. Tinker, John F., "Imprecision in Indexing--Part II," American Documentation, (XIX, July 1968), 322-330.

4. Tinker, John F., "Imprecision in Meaning Measured by Inconsistency of Indexing," American Documentation, (XVII, April 1966), 96-102.

5. Voigt, Melvin J., and Joseph H. Treyz, "The New Campuses Program," Library Journal, (XC, May 15, 1965), 2204-2208.

3. DESIGN OF FILE STRUCTURES FOR ON-LINE BIBLIOGRAPHIC CONTROL SYSTEMS

by Jorge Rodriguez

3.1 Introduction

3.1.1 Design Requirements for Information Retrieval Systems

Designing an information retrieval system for a library is a difficult task. The intricate complexity of the decisions and the large dimension of the system that is required contribute to the present inability of performing a unified and simultaneous analysis of all its components. Furthermore, the design of such a system requires the effort of specialists who are qualified to solve specific problems related to only certain aspects of the total project.

In general, we can group the design requirements of an information retrieval system as follows:

a. Provide the Most Useful and Necessary Services related to the use of libraries. More specifically, to facilitate search of information relevant to library users, and to expand the kinds of library services into areas which are projected as necessary for the future.

b. Minimize Retrieval Time. A good measure of the usefulness of any retrieval system is the time saving that can be provided to the users. Also, if the retrieval time in the machine system is larger than the time required to locate a specific record using index cards, then the retrieval system becomes unfeasible. Consequently, this parameter is most important for our specific system.

c. Minimize Storage Requirements. Decisions that directly or indirectly affect the structure or organization of the files will affect the amount of storage needed. Associated with the storage requirements are many costs which should be minimized wherever possible.

d. Provide for Efficient File Updating. Two of these processes are of importance in bibliographic application.

(1) Additions. This represents an important process in library systems due to the high rate of new publications that can be expected.

(2) Changes. This is mainly concerned with corrections of errors in stored records, which are impossible to eliminate completely.

e. Maximize File Security. It is desirable to minimize the possibilities of destroying any information in the files due to malfunctioning of the system or erroneous updating procedures.

Any information search and retrieval system will contain the following phases:

a. Input Specification Phase. In our system this process will be performed at some kind of terminal where the requests will be specified and placed by the file users.

b. Execution Phase. This phase has three sub-parts:

(1) Compilation: the transformation of the request topic into machine language readily usable by the central processing unit.

(2) Retrieval Phase: includes the search for the requested records and the retrieval of the lists of master records, if any, to the central processing unit.

(3) Analytical Phase: information derived from the content of the files is processed for logical conditions in order to present only that information which the user has requested. Also, this information is transformed into a language understandable to the user.

c. Output Phase. Finally, the information requested is sequenced, formatted, and presented at some kind of display terminal.

The time required to input, compile, analyze, and output will not be affected by the structure and organization of the files. For this reason, the present analysis is limited to the Retrieval Phase only.

3.1.2 Objective of File Structures Analysis Task

Of the general objectives mentioned, only b and c (minimization of retrieval time and storage) are related to parameters that can be expressed readily in mathematical form. Hence, this task will be concerned mainly with the optimization of the Retrieval Phase, using the criteria of minimizing costs associated with both auxiliary (secondary) storage and total processing time.

3.2 General Model of a File System

This part will be directed to the design of a generalized model of a bibliographic file management system which will be useful in the analyses to follow.

This model can also be adopted for simulation purposes because some of the essential features desirable for simulation are present.

This study will be mainly analytical and specialized on structures determined by the type of storage devices under consideration and the characteristics of bibliographic catalog data. However, the concept of the general model will always be implicit in each structure. Also, this model will prove useful as an analytical tool in the event that new devices and organizations appear worth considering in the future.

3.2.1 Fundamental System Components

In general, we can state that all retrieval systems are composed of two types of components which will be referred to as Storage Blocks and Information Linkages.

The storage blocks are associated with a physical storage unit. The content of these blocks can differ considerably; however, all the possible kinds of blocks are formed by a combination of only four different types of data.

The information linkages are associated with flow of information that is controlled by a central processing unit. Hence, considerable simplifications are also achieved by a proper classification of these linkages.

3.2.1.1 Storage Blocks

A Storage Block (SB) is a structural entity which is operationally defined as the basic logical design unit of a file. It should not be confused with blocking as an IOCS programming concept synonymous with "physical records," or with bucket, a term for an addressable location in storage having a specified capacity in terms of data records which can be stored at the location or its extensions, e.g., a cylinder or a track in a disk pack.

The four different forms of content that can singly or in combination form a storage block are: a) device address, b) keys, c) stored information such as catalog records, and d) pointers (or links). Both a and b can be regarded as Access Points and c and d as Data.

Under certain circumstances the content of an SB may be identical to more than one of the above forms. For instance, it is possible to structure a file such that a list of author names (an index file) could serve as both search keys and information elements. The authors would not be repeated in the master record for the file with which they were associated but would be linked back by a pointer to the index file. In another case, the author names might be treated as both keys and addresses. The internal representation of the name information could serve as the input to some algorithm which transforms it to a storage address. In a further case the record number (primary key) could serve as the file address ("key-as-address" techniques). This overlapping of forms does not present any difficulty for our analysis, however.

The reason for defining storage block in the way we have is to provide for analytically fruitful manipulations of the structure of a file.

These operations involve alternative techniques of record and file segmentation - splitting logical records or logical files into multiple physical files or regrouping records into useful sub-files by attributes such as publication date.

The most important feature of a Direct Access Storage Device (DASD) is its addressability to any track and information within the track by pointing directly (non-serially) to its location. This means that a unit of information possesses a physical address just by being stored, by action of the system. The 2314 Direct Access Storage Facility disk, for example, contains in each track a field called the Home Address which defines the physical location of the start of the usable storage area to facilitate operation of the access mechanism. Because storage blocks, no matter what their form of content (keys, information records, or pointers) are inherently thus addressable, we will not include the address in describing the alternative block forms which will be postulated, in order to eliminate confusion. All blocks are assumed to be addressable, either directly by actual device address or indirectly by various kinds of relative address. Actual addresses will not be considered in the File Organization Project since they have the disadvantage of rendering the data set unmovable. A further reason for eliminating the address from the analysis is that it simplifies the storage allocation space (except when explicitly recorded as data, i.e., an index reference, called a pointer in this analysis).

There are a large number of conceivable combinations of forms in storage blocks. We have selected a limited number of these for examination as feasible alternatives. After enumerating the blocks which seem most pertinent for analysis, we will describe the concept of information linkages briefly and then pass to a review of the block structure alternatives to see which ones are feasible for extended analysis.

There are six possible combinations of blocks which could be used to build a direct access file. These are listed in Figure 12.

FIG. 12:
TYPES OF STORAGE BLOCKS (SB's)

Block Type	Form A	Form B	Form C
I.	Information	-	-
II.	Information	Pointer(s)	-
III.	Pointer(s)	-	-
IV.	Key*	Pointer(s)	-
V.	Key	Information	-
VI.	Key	Information	Pointer(s)

Note that the access path to a block always connects at Form A in any given type of block. Content type (a), address, is always implicit as the location of the block. That is, Form A is the part that is addressable from outside the block. It is necessary again to point out that the storage block concept is distinguishable from blocking factor. A storage block might be co-extensive with a physical record made up of many logical records. In that regard, an SB can be said to exhibit the property of having a "blocking factor." However, as a logical entity, a single SB may extend across multiple physical records, or there could be several SB's in a single physical record. And a logical record might span multiple linked storage blocks, stored on different types of devices, as in a horizontally segmented file. "Blocking factor" is restricted to the programming sense of a unit physical record on a storage medium, consisting of one or more logical records, separated by inter-block gaps on tape. It has an operational meaning that is important to remember; that is, as the unit of data transmitted between main storage and an I/O device by the manufacturer supported operating system data management facilities, when the direct access method is used.

3.2.1.2 Information Linkages

Important to a file system where the structure is complicated by efforts to optimize such parameters as inquiry response time is the notion of information linkages. When a file segmentation is undertaken in order to improve retrieval time, as when the expected volume of search or update activity is unevenly distributed across file records, the transfers of search requests, intermediate information, and output products to the requester may involve complex paths to and from the CPU and secondary storage, and within the secondary unit. To account for costs of such transfer paths, the concept of information linkages was devised. Such linkages are associated with the flow of information to and from users and files, which may be in the form of input requests for searches, requests for I/O operations on data sets issued by the search program in the CPU to the peripheral file unit, and retrieved information being sent for display to the requesting individual. Using these types of flows, the linkages can be grouped into three general categories to determine their effect on file organization. Fig. 13 shows these categories and the possible sub-types of linkages under each category.

---

*'Key' in Form A does not exclude the possibility of placing more than one attribute in a keyed record in what is called a combined secondary index file. An example would be a key combining author plus some portion of title fields.

FIG. 13:
TYPES OF INFORMATION LINKAGES

Linkage Type	Origin	Destination
I. Message Block to Storage Block	Input Message	Key (SB Types IV, V, or VI*)
II. Storage Block to Storage Block	Pointer (Form A, B, or C*)	Key
	Pointer (Form A, B, or C*)	Information (Form A* SB's only)
III. Storage Block to Message Block	Key	Output Message
	Information	Output Message

The forms of content which can be included in the various Message Blocks - an Input Block, e.g., a command language repertory, search arguments involving a file attribute code set, such as 'AU' for Author index file and a key value such as the name 'SMITH', and an Output Block, e.g., the message repertory and action options displayed on a visual terminal to the requester, are beyond the scope of the present effort and will be dealt with in other sections of the report.

Since these linkages all imply hardware and software actions (seeks, reads, etc.) there are definite time penalties implied in file structures involving numerous transfers among message blocks and linkage segments (storage blocks). These timing factors will be examined in detail in subsequent discussion.

---

* See Fig. 12.

3.2.2 Feasible Storage Blocks

Continuing the analysis, we now introduce the characteristics of bioliographic records and the types of devices and I/O techniques that are presently available, the effect of which is to limit the number of feasible storage blocks. The present analysis will include only IBM direct access storage devices. As a result, the following storage blocks previously defined are not feasible for secondary key access:

FIG. 14:
UNFEASIBLE BLOCKS

SB Type	Forms in Block		
V	Key	Information	
VI	Key	Information	Pointers

These are ruled out because of the requirement for multiple secondary key access to bibliographic files - authors, titles, subjects, etc. In the blocks containing index structures, one would not wish, therefore, to repeat the bibliographic master record data (the "information" form). The argument could be made for using Blocks V or VI for at least the primary key (record number), and indeed this should be considered.* For alphanumeric keys, however, the Direct Organization, which supports variable-length records, does not possess the part-key feature which is manufacturer-supported in the indexed sequential access method software. And the ISAM software available to the ILR facility does not currently support variable-length records. Hence, the storage blocks that will be considered in the remainder of the analysis are types I-IV, shown in Fig. 12. The list of possible linkages remains as in Fig. 13.

---

*The master file record numbers in the ILR facility have been stored in a "finder file" of master record link addresses because the links once computed are used frequently in the index file-building process. A short, fixed length file of these pointers was desired for efficiency.

3.3 Single-Device File Organization

3.3.1 Purpose of Analysis

The present analysis is intended to optimize the organization of bibliographic records within one direct access storage facility of a given type (which may contain multiple storage units, e.g., disk packs), under two criteria:

a. Minimum Average Retrieval Time, as the principal measure of the retrieval response time distribution, and

b. Minimum Cost Associated with Auxiliary Storage and Processing Time. Future work will extend the analysis to organizations using multiple facilities of different types (e.g., disk facility and data cell facility).

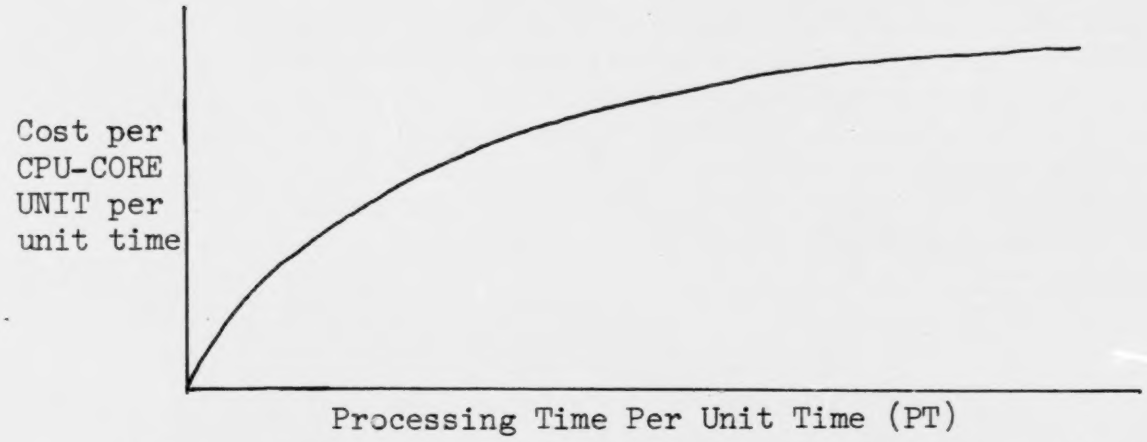
3.3.2 Costs

The following costs will be assumed given:

a. Cost of Auxiliary Storage (Cs). This refers to the cost of DASD space which is assumed to vary linearly with respect to the number of bytes stored.

b. Cost of Processing Time (PT). This represents the cost of using the central processing unit (CPU), and it is expected to be a concave, non-decreasing function as shown in Fig. 15.

FIG. 15:
PROCESSING COST CURVE



Generally, the cost of processing time of a central processing unit depends also on the amount of main storage that is used. For this reason the cost is shown per core unit, which represents the amount of core used in the processing of a request. For this analysis it will be assumed that the central processing unit is used for costing purposes all the time during the search and retrieval stage.

3.3.3 Assumptions

The following assumptions are also adopted in this analysis.

Assumption 1: All the unkeyed files are assumed to be blocked to full track capacity where possible.

Assumption 2: The seek time is independent of the record length and its location. This is justified considering that the bibliographic records are shorter than the track length of the first DAS facility to be considered. Therefore, the probability of a record being split into two tracks is low.

Assumption 3: For an indexed file of fixed number of index levels,* the amount of storage required is assumed proportional to the number of keys. In a blocked unkeyed file, the storage requirements are equal to the total sum of the records' lengths, but in an index file the storage requirement is more than the total lengths of the records due to the overhead storage needed for cylinder indexes and track indexes (ISAM). Then, this assumption states that the extra amount of storage needed is proportional to the number of keys. A useful implication of this assumption is that for a given keyed file, we can obtain the cost of storage per keyed record in a given device.

Assumption 4: The requests are served on a first-come-first-served (FIFO) basis, and the requests of each record are randomly distributed in time. This implies that the system cannot expect more than one request of a record at a given time.

Assumption 5: The structure and organization of the file is to be regarded as having no effect on the user's behavior. This implies that the cost and retrieval time variations will not affect the usage frequency of any record.

The catalog structure that will be studied contains several index files; i.e., author, title, subject, etc., and the correspondence of the keys and records is not unitary (one-to-one).

These different types of indexed files are assumed to be the only types of on-line accesses to the files. Non-indexed on-line access to the content of master records is a separate analytic issue outside the scope of the present report.

---

* Some useful performance analyses of the manufacturer-supported indexed sequential access method software are now becoming available. For example, in regard to trade-offs involved in going to a master-level index, and other aspects of ISAM, see Lum, V.Y., H. Ling, and M.E. Senko, "Analysis of a Complex Data Management Access Method by Simulation Modelling," In AFIPS Conf. Proc., Vol. 37, (Fall Joint Comp. Conf., 1970), Montvale, N.J.: AFIPS Press, 1970, 211-222.

Assumption 6: The number of records associated with a key is independent of its usage frequency.

Assumption 7: The index key field length is a fixed value. (It is selected as a file parameter at load time, up to a maximum specified in the manufacturer-supported software.)

More assumptions will be adopted at later stages of the analysis.

3.3.4 Analysis of Alternative Structure Concepts

The present problem represents a special case of file structure which limits the alternatives to only a few cases. For this reason, each organization will be described and a method to select the best one will be provided.

The type of index file that will be considered for this system is the Index Sequential Organization for DASD, which represents the most attractive organization method because of the key prefix retrieval capability in the manufacturer-supported access method. This feature permits any number of characters to be submitted in the search key to be matched against the index key. This is called the "part-key" facility.

3.3.4.1 Unfeasible Structures

We begin the discussion of alternative structures comprised of Type I-IV Storage Blocks with a review of file structures which can be rejected on various grounds.

Most of the possible structures will be ruled out due to the particular characteristics of bibliographic catalogs and to the timing characteristics of DAS devices. The following is a discussion of the unfeasibility of the structures shown in Fig. 16. In the course of this discussion, we will point out other factors related to the dynamics of the files.

a. Threaded List Master Records. Structure 1 represents a conventional linked list. The key "directory" is placed in a separate index file. The master records associated with an index key are retrieved in a sequential pattern, where each record contains a link address or pointer to the next associated master record to be retrieved. However, each of the various keys associated with a given master record is not necessarily associated with the same group of master records. This will thus require in each master record a large number of pointers associated with different keys and records, which will be extremely difficult to maintain. Also, it rules out the possibility of blocking the master file to full track capacity because storage has to be provided for additions to the chains. Furthermore, its average retrieval time will probably

---

*When multiple secondary key access is required, as in bibliographic systems, this structure is called a multiple threaded list, or Multi-List Structure.

be higher than that of more feasible structures to be discussed.* For this reason, we now consider various forms of inverted list structure; that is, where the links to master records are physically removed from the master file.

b. Segmented Master File. In connection with this structure, it is appropriate to review the role of file and record segmentation in this analysis. Structure 2 has two possible refinements, based on activity of the file. One is called "horizontal segmentation," and the other "vertical segmentation." In horizontal segmentation, the most frequently used parts of each of the master records are placed in the first or primary segment, the one which is linked by pointer from the access (index) file, and the remainder of the record is placed in segment 2. The two physical segments (files) form one logical master file. A pointer links the record segments in each of the two sub-files.

In vertical segmentation, some variations are possible. Here, whole records are separated into relative high use vs. low use groups and allocated to segment 1 and segment 2 accordingly. Two possible linkage arrangements are possible: the first is to use two sets of pointers in the index file, one set pointing to the high use segments and the other set pointing to the associated low use records. The other possibility is to connect the segment 2 group of master records by a pointer from the trailing record in segment 1 of the master file, as shown in Fig. 16. This is a variation on Structure 1, and has the same disadvantages.

These structures can be improved by employing multiple devices where feasible. For example, the high use segment could be allocated to a fast but expensive DASD, while the lower use records are stored on a slower but cheaper device, thus bringing optimization from the viewpoint of access time in relation to expected file activity.

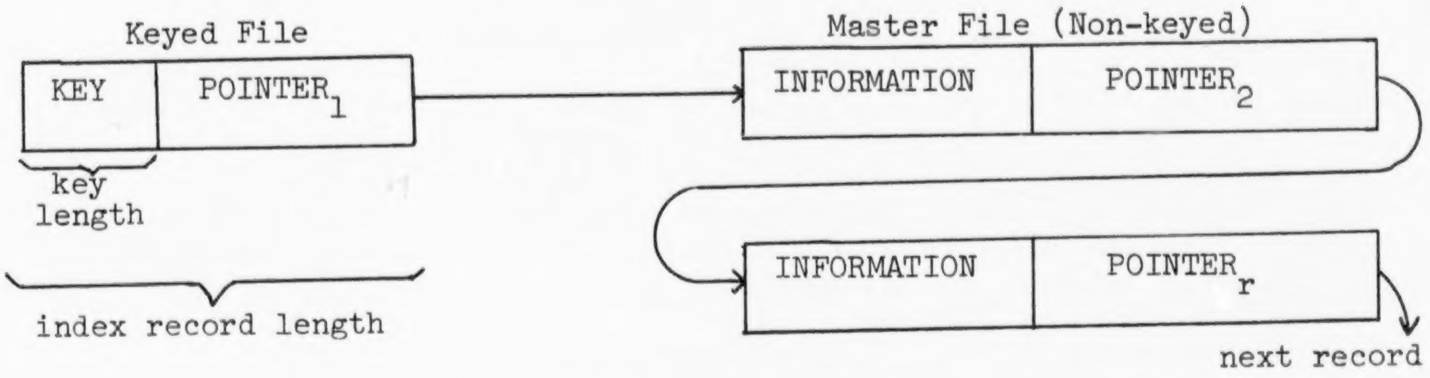
Combinations can be visualized. For example, vertical segmenting could be performed first on the master file, then secondary horizontal segmenting on the resultant high use (segment 1) group, in addition. This would leave only the high use sub-record portions of the high use group of records to be stored on the faster DASD, if that were found to be an effective scheme. Moreover, the same segmenting strategies can be performed on the index file. We turn our attention to that issue in the next section on "Feasible Structures."

---

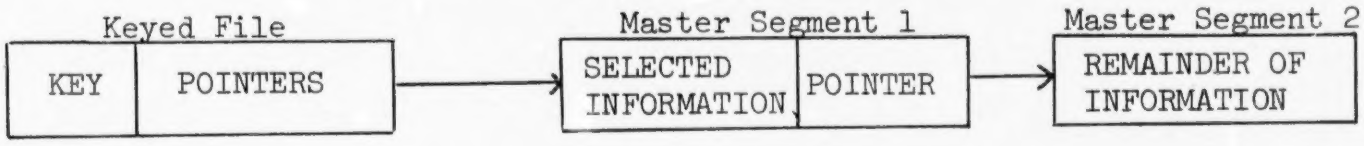
*For comparative performance analyses of threaded list structures vs. inverted lists, see: Lefkovitz, David, File Structures for On-Line Systems, New York: Spartan, 1961; Lowe, Thomas C., Design Principles for an On-Line Information Retrieval System, (Moore School Report No. 67-14), Philadelphia: Moore School of Elec. Engr., Univ. of Penn., Dec. 1966, or the same author's "The Influence of Data-Base Characteristics and Usage on Direct-Access File Organization," JACM (15, October, 1968), 535-548; and Martin, Lawrence David, A Model for File Structure Determination for Large On-Line Data Files, Pullman: Systems Div., Washington State Univ., 1968.

FIG. 16:
UNFEASIBLE STRUCTURES

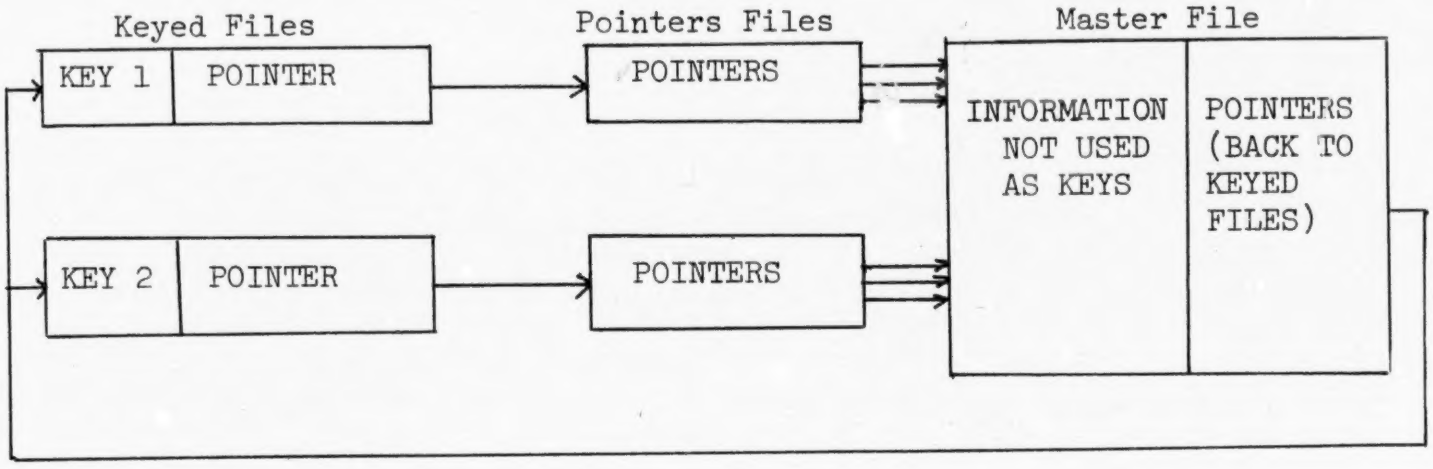
STRUCTURE 1 - THREADED LISTS


Keyed File
KEY
POINTER$_1$
key length
index record length
Master File (Non-keyed)
INFORMATION
POINTER$_2$
INFORMATION
POINTER$_r$
next record

STRUCTURE 2 - MASTER FILE SEGMENTS


Keyed File
KEY
POINTERS
Master Segment 1
SELECTED INFORMATION
POINTER
Master Segment 2
REMAINDER OF INFORMATION

STRUCTURE 3 - CIRCULAR LINKED LISTS


Keyed Files
KEY 1
POINTER
KEY 2
POINTER
Pointers Files
POINTERS
POINTERS
Master File
INFORMATION NOT USED AS KEYS
POINTERS (BACK TO KEYED FILES)

Structure 2 may be justified, as will be shown later, when some costs can be reduced by storing information of infrequent use in slower, more inexpensive devices, or when the retrieval time is affected considerably by the length of the information to be read. Acceptance of Assumption 2 eliminates that time factor for the present stage of analysis.

c. Circular Linked Lists (Non-Repeated Keys). Structure 3 is an attempt to save some space by excluding from the master file the fields or terms that are used as keys; this would be useful if these terms were of fixed length. However, this is not the case with bibliographic data. The keyed terms are originally of variable length but are truncated into fixed length fields in the index record. During this process, some information is lost, and in some cases this key is of little value when the keyed terms themselves constitute the information requested by the searcher. Moreover, even if variable-length key fields for index records were supported in the software, there is often a representation problem. For example, a key value such as an author name which would be stored in upper and lower case in the master record, may have to be stored in upper case, with diacritics, etc., removed, in the index key for convenience in searching.

Moreover, there is a further space penalty in chaining from each key value instance in a master record back to the index file. A retrieval and presentation problem also lies in the method used to link the key values and master record for display purposes.

3.3.4.2 Feasible Structures

a. Inverted Lists (Repeated Key, Non-Segmented Master File). The only types of structures that are considered currently feasible are shown diagramatically in Fig. 17. These are special cases of inverted lists. Separate analyses will discuss the possibilities of segmenting a single keyed file into linked physical files according to criteria such as key length. That analysis is independent of the kind of structure to be selected, for it is performed for each keyed file individually. The present analysis is also performed for each type of key individually because the different keyed files function independently.

b. Structure 4 - Two-Level Linked Lists with Repeated Keys. Structure 4 represents a linked-pointer index concept intended to overcome a system limitation in the access file, which must be fixed record length format. It includes a field for the key and a field for a single pointer, which points to the head of a list stored in an intermediate file called the Pointers File or Address File. The pointers in the Address File are link addresses of master records associated with the key value.

c. Structure 5 - Unit Lists. Structure 5 also involves keys which are copied out of the master file. It can be implemented with either fixed or variable length index records, and the intermediate file is not used. 5a places the key and all master file pointers together in a fixed length record. Structure 5b represents the more conventional inverted list structure. 5b is a relaxed form of Structure 4 in that variable length records are permitted in the access file, thereby obviating the necessity for a Pointers File. The key field must still be fixed length.

At the time of this writing, only Structure 4 and Structure 5a or slight modifications could actually be given trial implementation in the ILR facility, due to software limitations. In fact, the difference between the storage requirements of Structure 4 and Structure 5a is due to a (temporary) constraint on implementation which requires fixed length records in keyed files.* This imposes the condition in Structure 5a of setting the record length to a specified maximum value at load time for the file. That value must therefore correspond to the sum of the maximum desired index key field length plus the length of the field containing the pointers. The pointers field must be long enough to contain all the pointers for that key instance which is associated with the largest number of master records. As a consequence, most of the index records will contain unused space. (The key field length must also be set at some predetermined fixed length by the file designer.)

3.3.4.3 Index File Segmentation Based on Index Record Length

To overcome this waste of storage, the records of an index file in Structure 5a can be grouped according to their total record length, or equivalently according to the number of master records, i.e., pointers, associated with their keys. This is to be distinguished from segmenting the file on the basis of key field length. Using structure 5a as a basis, in the following section we will analyze the effect of dividing the access file according to record length. The segmented Structure 5a will require the same amount of storage as that of Structure 4, which has no storage wasted for pointers. However, each division of the 5a index file represents an increase in retrieval time due to the time to transfer the search from one sub-file to another.

The segmented structure 5a requires less access time than that of Structure 4 because of the absence of the intermediate pointers file, but the storage requirements are longer due to the allowance for the key having the greatest number of pointers. The times which are of importance in the process of search and retrieval are:

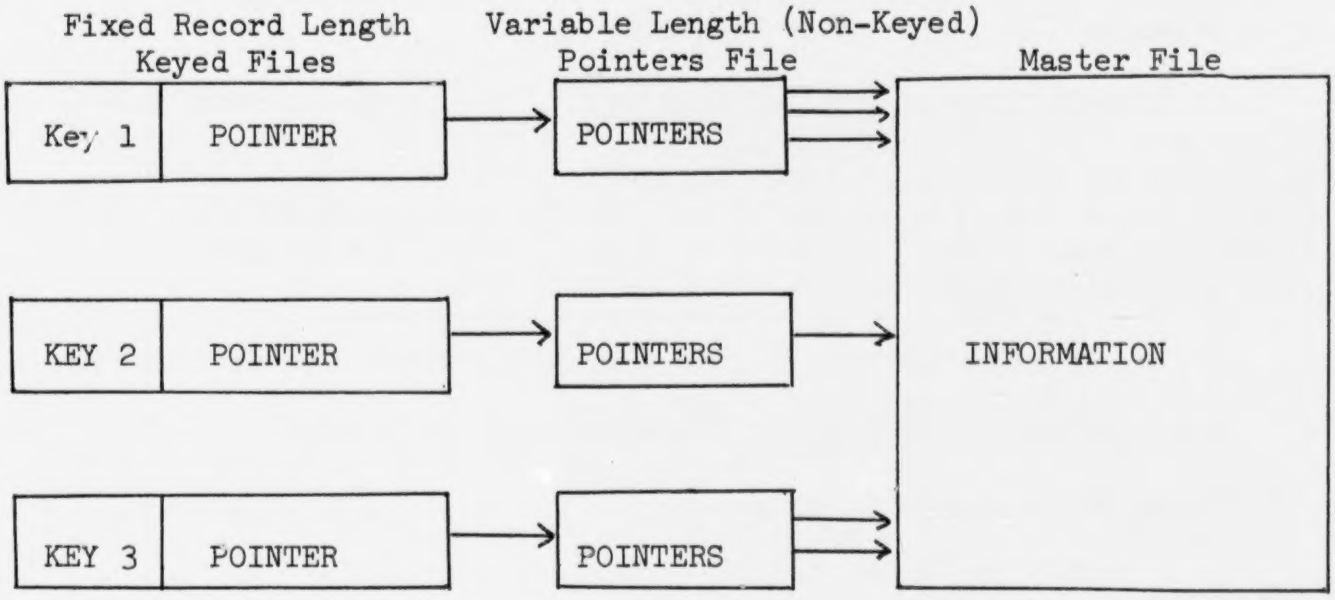
a. Non-Direct Access Device Timing Factors. Time required by the machine to interpret the commands and prepare and transfer the instructions to the control unit (CU) of the DASD. It includes search program processing time, access method processing

---

*A limitation entirely attributable to the fact that variable length format records are not supported by IBM for the Indexed Sequential Access Method in Releases of OS up to Release 17.

FIG. 17:
FEASIBLE STRUCTURES

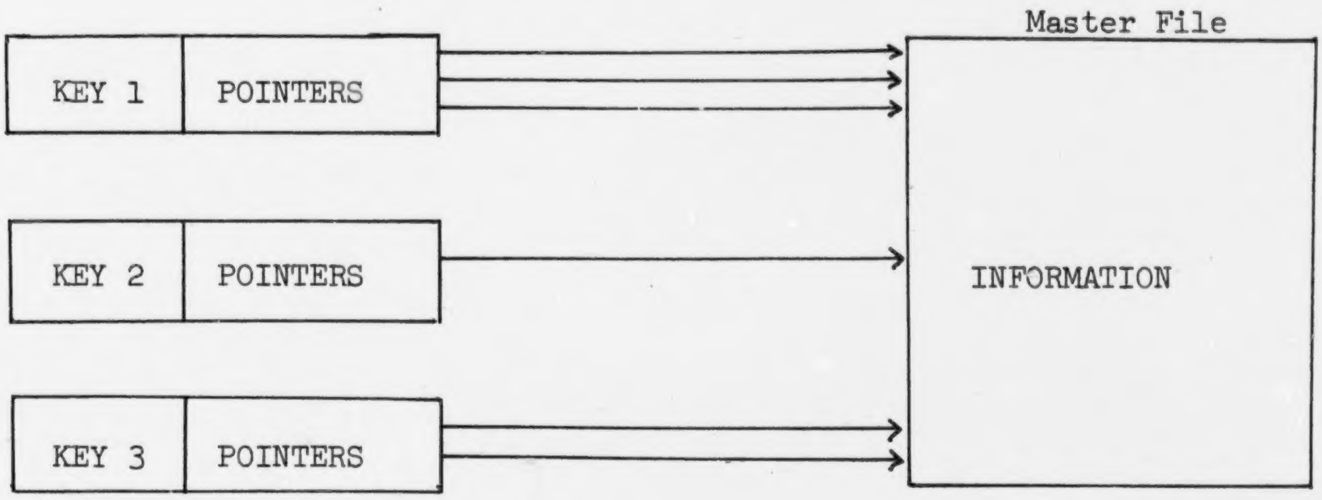
STRUCTURE 4 - TWO LEVEL LISTS (REPEATED KEYS)


Fixed Record Length
Keyed Files
Variable Length (Non-Keyed)
Pointers File
Master File
Key 1
POINTER
POINTERS
KEY 2
POINTER
POINTERS
INFORMATION
KEY 3
POINTER
POINTERS

STRUCTURE 5 - UNIT LISTS

5a. Fixed Record Length Index Record

5b. Variable Record Length Index Record


Master File
KEY 1
POINTERS
KEY 2
POINTERS
INFORMATION
KEY 3
POINTERS

time and operating system supervisor time. It is valid to assume this time as constant for a given type of file. Hence, these will be represented by Tk for a keyed file and Tunk for an unkeyed file, where Td>Tunk.

b. Seek Time (T_B). Also called Access Motion Time. This is the time required to reach any location of the DASD. As mentioned before, this is a constant for these types of devices. For our analyses, it includes two other parameters, rotational delay (latency) and data transfer time, which are constants.

c. Search Time in a Keyed File (K). The average search time in a keyed file under Index Sequential Organization can be estimated by adding the times to go from one index level to another, and the average times required to search each index level sequentially. Before estimating this time, the number of index levels in a given index file and their respective lengths should be obtained.

Hence, the average search time K in an indexed file is given by:

$$K = T_B(NIL-1) + \frac{ROT}{2}[MT1 + MT2 + MT3 + CT + TI + 1] \qquad (1)$$

$$\text{where } MT1 = \min[NTM, M1] \qquad (1.a)$$

$$MT2 = \min[NTM, M2] \qquad (1.b)$$

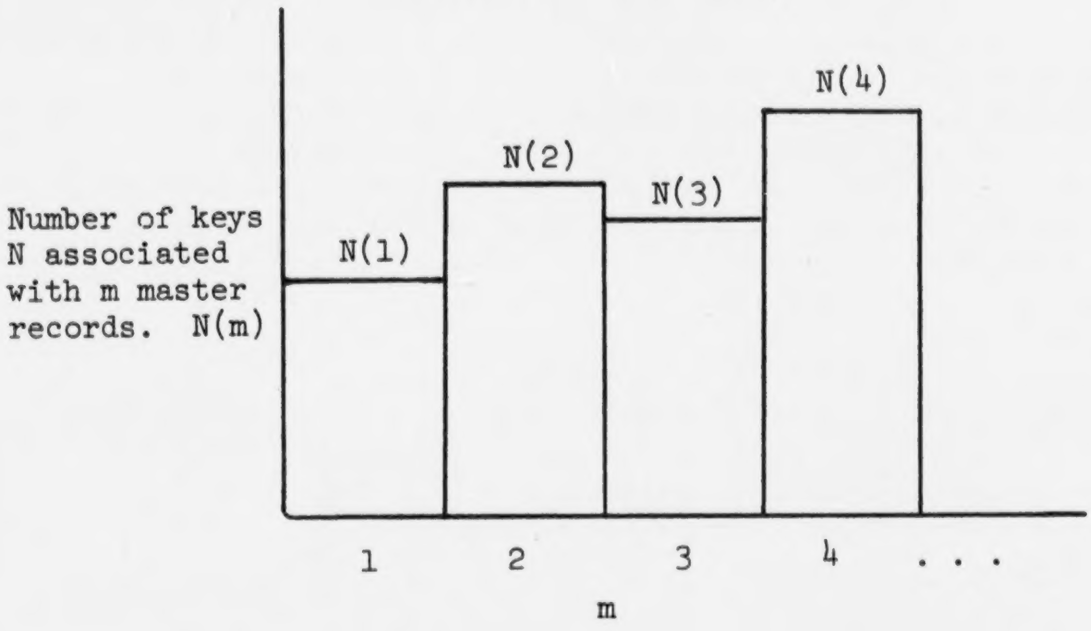
$$MT3 = \min[NTM, M3] \qquad (1.c)$$

$$CT = \min[NTM, CIT] \qquad (1.d)$$

NIL = number of index levels (system created)
ROT = time required to read a track (equal to one full rotation in a disk type of DASD). Includes Transfer Time.
M1 = number of tracks in first master-level index
M2 = number of tracks in second master-level index
M3 = number of tracks in third master-level index (if required)
CIT = number of Cylinder Index tracks
TI = number of Track Index tracks
NTM = minimum number of tracks in an index level that will justify a higher level.

For this analysis, the distribution of Fig. 18 is assumed given:

FIG. 18:
DISTRIBUTION OF THE NUMBER OF RECORDS
ASSOCIATED TO EACH KEY



Also, for each keyed file, the following data is assumed given:

n = total number of keys
Sk = total number of requests per unit time, which is equal to the number of key searches.
Sp = total number of successful searches; this is equal to the number of accesses to the Pointers File, under structure 4.
Sm = total number of records retrieved; this is equal to the number of accesses to the master file.

With this information given, the total processing time can be evaluated for each structure.

For structure 4, we have:

Index File:

$$\text{Tindex} = \text{Sk}(\text{Tk} + T_B + K) \qquad (2)$$

Pointers File:

$$\text{Tpointer} = \text{Sp}(\text{Tunk} + T_B) \qquad (3)$$

Master File:

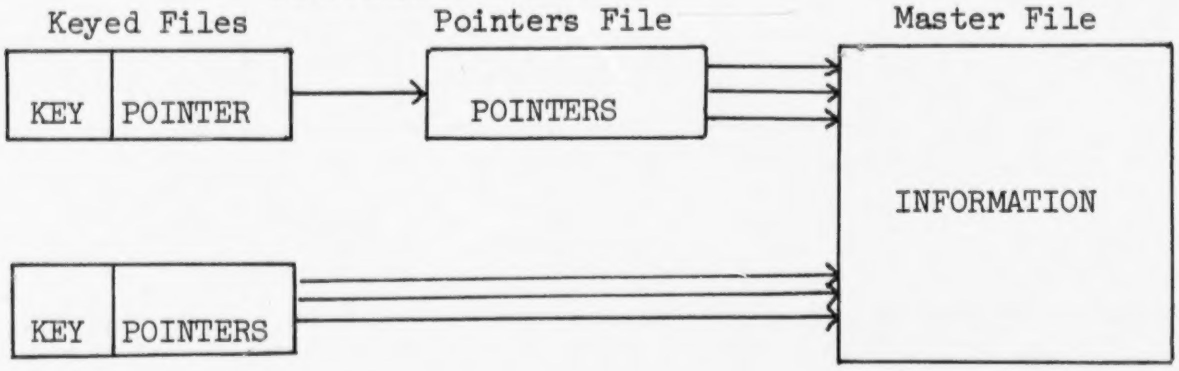
$$\text{Tmaster} = \text{Sm}(\text{Tunk} + T_B) \qquad (4)$$

Therefore, the total time for structure 4 is:

$$\text{Tst.4} = \text{Sk}(\text{Tk} + T_B + K) + (\text{Sp} + \text{Sm})(\text{Tunk} + T_B) \qquad (5)$$

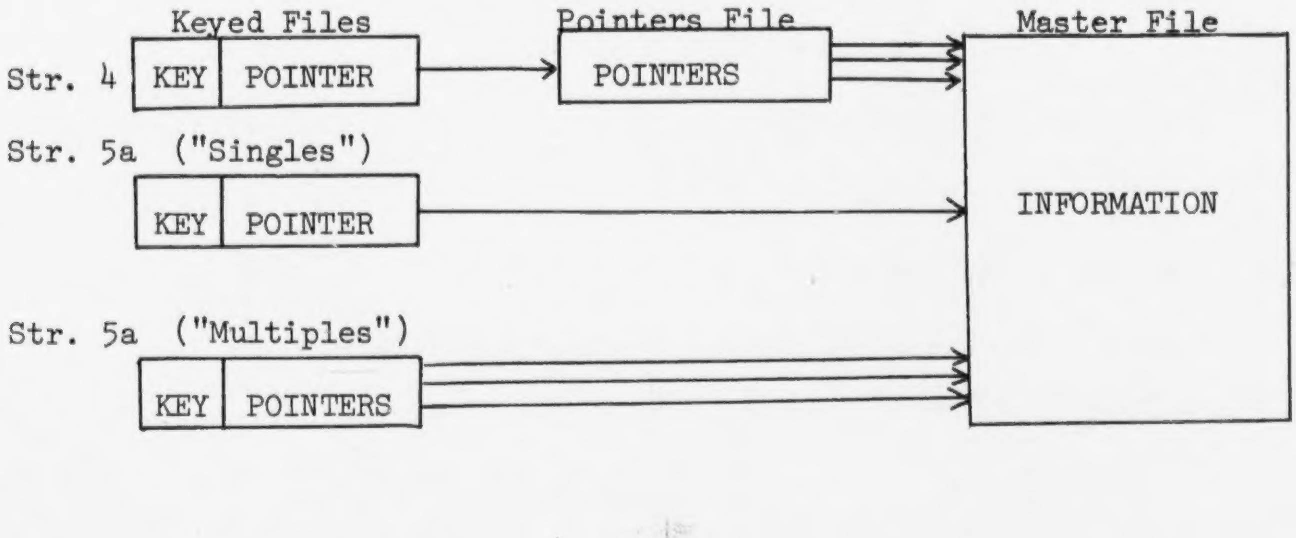
The number of groups (m) of the distribution shown in Fig. 18 will be fairly large for the bibliographic catalog under consideration; consequently, the adoption of the segmented Structure 5a will require much larger processing time than that of Structure 4, and no storage savings can be accomplished. For this reason a more useful analysis will now be directed to the effects of using a mixed structure, called Structure 6, wherein some groups of index records are organized together under each of Structures 4 and 5a. Two discrete files, each organized under Index Sequential, are necessary. A diagram of a mixed structure is shown in Fig. 19. This is designated Structure 6 and is the form used for analysis in the remainder of the paper.

FIG. 19:
STRUCTURE 6 ("MIXED")


Keyed Files
Pointers File
Master File
KEY
POINTER
POINTERS
INFORMATION
KEY
POINTERS

In a preliminary analysis of the Santa Cruz data base, it was determined that the second largest group constituted approximately 6% of the index records for the author names file. Index records in this group had only 1 pointer, i.e., were associated with only one master record). These unitary keyed records constitute a special case of Structure 5a. When this occurs, a variant of Structure 6 can be implemented immediately prior to further analysis of the key/master record ratios, for division by record length. The single-pointer index records can be retained in the Index Sequential File for Structure 4, using the same record length as for Structure 4. The structure in this case is as follows:

FIG. 20:
STRUCTURE 7 ("MODIFIED MIXED")


Keyed Files
Pointers File
Master File
Str. 4
KEY
POINTER
POINTERS
Str. 5a ("Singles")
KEY
POINTER
INFORMATION
Str. 5a ("Multiples")
KEY
POINTERS

The group of Structure 4 index records (not in the M=1 portion of segment 1) is, of course, still subject to division according to the procedures suggested for creating a mixed structure based on index record length. The "singles", being already of minimum record length, would thus be excluded from candidacy for segment 2. If singles constitute the largest group, the division procedure can proceed with the next largest group, depending on the conditions.

3.3.5 Evaluation of Mixed Structure

The problem now is to determine the conditions that will justify the organization of a given group of index records under Structure 5a, as part of a mixed structure. The mixed structure (Structure 6) will be the subject of the remainder of this analysis.

From the distribution of Fig. 18, select the largest group. Let this be N(M) and organize it as an ISAM file using the minimum record length for M pointers under structure 5a. The remaining records are organized under Structure 4, using a record length for one pointer (to the Pointers File). The next step is to evaluate the total processing time for the mixed structure.

Because of Assumption 6, the total number of requests to each index file depends only on the number of records in each file. Also, the time to transfer from one index file segment to another will be relatively large. Consequently, it is more efficient to search the largest file segment first.

On this basis, the Total Processing Time will be evaluated for two cases of Structure 6 (Mixed Structure):

3.3.5.1 Case 1: $N(M)<n-N(M)$. Largest Group of Keys Less Than 50% of File

Processing time for the first segment of the index file:
(Structure 4) (Largest File)

$$\text{Tindex } 1 = Sk(Tk + T_B + K1) \tag{6}$$

For the second segment of the index file:
(Structure 5)

$$\text{Tindex } 2 = [Sk - Sp(1-\frac{N(M)}{n})]\,(Tk + T_B + K2) \tag{7}$$

For the Pointers File:

$$\text{Tpointer} = Sp(\frac{n-N(M)}{n})\,(Tunk + T_B) \tag{8}$$

and, for the master file, the same as (5). Hence, the total time for this case is:

$$\text{Tmixed str. case } 1 = Sk(Tk+T_B+K_1) + [Sk-Sp(1-\frac{N(M)}{n})]\,(Tk+T_B+K_2)+ [Sp(1-\frac{N(M)}{n} +Sm]\,(Tunk + T_B) \tag{9}$$

where K1 is the average search time for the keyed file segment left under Structure 4, and K2 for the one under Structure 5.

Now it will be proven, as might be expected intuitively, that use of the mixed structure, when Case 1 obtains, is not justified for any value of N(M) in the designated range. It will be proven that the following is true:

$$\text{Tmixed Str. Case } 1 > T_{ST4} \text{ or Tmixed Str. Case } 1 - T_{ST4} > 0 \qquad (10)$$

After substituting (5) and (9) into (10) and simplifying, we get:

$$(\text{Tmixed Str. Case } 1 - T_{ST4}) = Sp(1-\frac{N(M)}{n}\ (Tk+T_B+K_2) + Sk(Tk+T_B) - \frac{N(M)}{n}\ Sp(T_B+Tunk) + Sk(K_1+K_2-K) \qquad (11)$$

The first term is clearly positive because $\frac{N(M)}{n}<1$, the second term is larger than the third one because Sp<Sk and Tunk<Tk, and the last term is also positive, for the search time does not decrease more than linearly as the number of records decreases. Hence, condition (10) holds.

3.3.5.2 Case 2: N(M)>n-N(M) Largest Group of Keys Greater Than 50% of File

For the first segment of the index file: (Structure 4)

$$\text{Tindex } 1 = Sk\ (Tk + T_B + K1) \qquad (12)$$

For the second index file: (Structure 5) (Largest file)

$$\text{Tindex } 2 = [Sk-Sp(\frac{N(M)}{n})]\ (Tk + T_B + K_2) \qquad (13)$$

The times for the pointers file and the master file will be the same as those for Case 1.

Hence, the total processing time for this case is:

$$\text{Tmixed Str. Case } 2 = Sk(Tk+T_B+K_2+K_1-K)-Sp(\frac{N(M)}{n})\ (Tk + 2T_B + K_1 + TUNK) \qquad (14)$$

Next, it will be shown that a mixed structure of Case 2 is justified for certain values of $\frac{N(M)}{n}$. Using equations (5) and (14) and simplifying, we get:

$$\text{Tmixed Str. Case } 2 - T_{ST4} = Sk(Tk+T_B+K_2+K_1-K)-Sp(\frac{N(M)}{n})\ (Tk + 2T_B + K_1 + TUNK) \qquad (15)$$

Expression (15) will be negative and the mixed structure of Case 2 is justified when the following is true:

$$\frac{N(M)}{n} > \frac{Sk(Tk+T_B+K_1+K_2-K)}{Sp(Tk+2T_B+K_1+TUNK)}$$

That is, the largest group N(M) which is the one placed in Structure 5a, and is searched first, must be a high enough proportion of the keys which are matches to the search requests of the system, in order to reduce sufficiently the search effort (time) spent on requests put to the group remaining under Structure 4. If the latter is too high, the mixed structure for this case will be inefficient.

This analysis has considered the effects on the retrieval time only, assuming that the storage requirements for the mixed structure are essentially the same as that for Structure 4.

The exact difference in storage can be estimated easily using the following form:

$$\text{Smixed ST} - S_{ST4} = (NT1 + NT2 - NT)ITC \qquad (16)$$

where NT = total number of tracks needed for the index file under Structure 4.

NT1 = total number of tracks needed for the index file under Structure 4 in the mixed structure.

NT2 = total number of tracks needed for the index file under Structure 5 in the mixed structure.

ITC = track capacity in bytes.

NT, NT1 and NT2 are given by program STOCAP* that was used to obtain the necessary parameters to estimate K1, K2, and K.

In the event that a mixed structure [illegible] Case 2 is favored, a similar analysis to the one just described can be performed to determine whether further subdivisions are justified. This analysis is performed using only the records that were left under Structure 4 after the first group's segment has been removed.

3.3.6 Segmentation of Index Files Based on Key Length

Now the possibility of splitting the files by key length, to achieve improved storage allocation, will be analyzed. As mentioned before, the terms in the records that are used as keys have to be transformed into fixed length fields not changeable after load time. In order to maintain a reasonable level of uniqueness in the key search, the key field length of a keyed file is usually set by a relatively small set of records having long keyed terms. As a result, many of the index records will contain a keyed field longer than necessary. The possibility of dividing the index file into groups having different key field lengths thus is plausible.

---

* A Fortran routine written to provide computational support in analyzing the model.

3.3.6.1 Objective of Analysis

At this point the following additional assumption will be adopted:

Assumption 7: The minimum key length required for an index record is independent of the key usage frequency and of the number of master records associated with it.

This assumption makes the analysis independent of the previous one. It can be performed independently on each keyed file determined by the previous analysis.

In this case the changes in storage requirement and processing time have to be considered simultaneously because both factors will change when a file is segmented. Furthermore, the effects of the partitions on these parameters are different.

Hence, the present analysis will not attempt to obtain an optimum, but to express in a useful way the effects of these divisions. In particular, this analysis will yield a graph of the minimum storage requirements versus the total processing time.

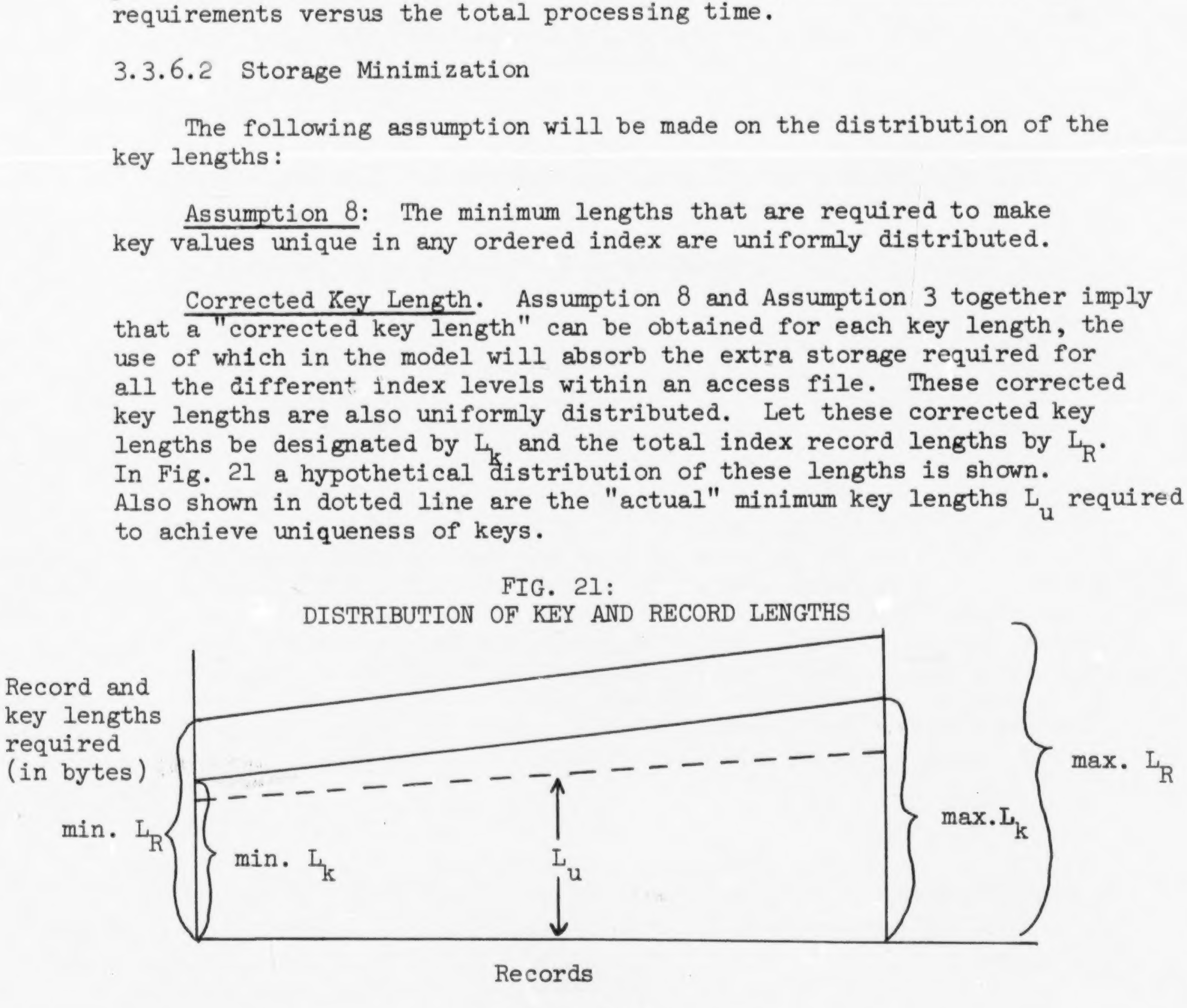
3.3.6.2 Storage Minimization

The following assumption will be made on the distribution of the key lengths:

Assumption 8: The minimum lengths that are required to make key values unique in any ordered index are uniformly distributed.

Corrected Key Length. Assumption 8 and Assumption 3 together imply that a "corrected key length" can be obtained for each key length, the use of which in the model will absorb the extra storage required for all the different index levels within an access file. These corrected key lengths are also uniformly distributed. Let these corrected key lengths be designated by L_k and the total index record lengths by L_R. In Fig. 21 a hypothetical distribution of these lengths is shown. Also shown in dotted line are the "actual" minimum key lengths L_u required to achieve uniqueness of keys.

FIG. 21:
DISTRIBUTION OF KEY AND RECORD LENGTHS



The minimum storage requirements for a given number of groups are achieved when the groups are all of the same size, and this minimum decreases as the number of groups increases. Hence, to obtain the graph of the minimum storage requirement versus total processing time, it is sufficient to evaluate these two parameters when the file is divided into different numbers of equal size groups.

The minimum storage required when the number of groups is h, is given by:

$$\text{Min. storage} = n(\min L_R) + \frac{n(h+1)}{2h} (\max L_R - \min L_R) \qquad (17)$$

where n = total number of records in the index file.
h = number of groups
$\min L_R$ = minimum record length (See Fig. 21)
$\max L_R$ = maximum record length (See Fig. 21)

And the time is given by:

$$\text{Processing time} = \sum_{k=1}^{k=h} [Sk - \frac{Sp}{h} (k-1)] (K_k+T_k+T_B) \qquad (18)$$

where K_k = average search time of the k^{th} keyed file which is estimated using program STOCAP and Expression (1). S_k, S_p, T_k, and T_B are as defined in the previous analysis.

Depending upon the severity of the increase in search time as the key length increases, it is clear that the files with shortest keys should be searched first, if the effect of usage frequency is not considered.

When the search is done sequentially, group-by-group in increasing key lengths, then by Assumption 8, the key length of the i^{th} group is given by:

$$L_{ki} = \min L_k + i \left(\frac{\max L_k - \min L_k}{h}\right) \qquad (19)$$

Again, the exact value for the storage can be estimated from the output of the program STOCAP using the form:

$$S = ITC \sum_{i=1}^{i=h} NT(i) \qquad (20)$$

where NT(i) is the number of tracks needed to store the i^{th} index file and ITC the number of bytes per track.

3.3.6.3 Processing Cost Minimization

Equations (17), (18), (19), and (20) give the value of the parameters using the criteria of minimum amount of storage. This, however, may not be a reasonable objective when cost of processing time varies significantly with changes in total processing time. For this last case a more representative graph will be that of Minimum Total Cost vs. Total Processing Time.

Here, the size of the groups will not be the same, and the analysis has to yield not only the minimum cost and total processing time, but also the size of each group, for a given number of groups.

The introduction of the processing cost eliminates the possibility of analyzing each index file independently. This is due to the nonlinearity of the graph in Fig. 15 and the fact that this cost is for the total processing time of the system and not for each file. Hence, this time is the sum of the processing times in each file, both index and master. The implication here is that the optimal partition of one file varies for different values of the total processing time, and because this time is a function of the structures of the other files, the overall dependence is established.

Consequently, any reasonable analysis will have to consider all the files simultaneously. To achieve this, an algorithm has been devised which generates the minimum cost vs. total processing time curve.

3.3.6.3.1 Design of Optimization Procedure

This algorithm describes an iterative procedure which starts with all the index files unsegmented, obtains the change in costs resulting by different divisions of each index file independently, selects the most advantageous change among all the files and possible partitions, changes the structure and repeats the process. Recall that the group which is removed from the basic corpus, by virtue of the division procedure, is formatted into Structure 5a (Unit Lists) and must have a fixed key length and a fixed record length specified at load time.

The total processing time is expected to increase as the number of partitions in each file increases. This is due to the time required to transfer the search from one segment to the next.

From the results already available on the distribution of the key length of the authors index file, it seems that in certain cases Assumption 8 is not justified. The algorithm will be essentially the same when Assumption 8 holds and when it does not. However, the process of evaluation of certain values will differ in each step. These differences will be pointed out when each step is discussed in detail.

3.3.6.3.2 Algorithm for Search Processing Cost/Total Processing Time Optimization

Step 1 - Estimate the storage requirements, total processing time, and total cost for the whole system without partitioning any index file with respect to key length, and plot their corresponding values in the minimum cost versus total processing time graph.

Step 2 - Obtain the optimal (minimum cost) group sizes for each index file when they are independently divided into h, h+1,..., h+ΔH groups, with respect to their key lengths, where h is the number of partitions of that file in the present structure.

Step 3 - Estimate the total cost when each of the index files is partitioned into h, h+1,...,h+ΔH group, without partitioning the others.

Step 4 - Select the best change of structure from the possibilities evaluated in step 3.

Step 5 - Plot the corresponding values of the structure selected by step 4 in the minimum total cost versus total processing time graph, and let this structure be the present solution for the next iteration.

Step 6 - Repeat step 2, 3, 4, and 5 recursively, until the minimum cost versus processing time graph achieves a trend where both parameters increase simultaneously.

3.3.6.3.3 Parameters

The following data are assumed given:

For the Keyed (Index) Files:

N(J) = number of records in the J^{th} keyed file

min Lk(J) = minimum key length for the J^{th} file

max Lk(J) = maximum key length for the J^{th} file

min L_R(J) = minimum record length for the J^{th} file

max L_R(J) - maximum record length for the J^{th} file

Sk(J) = total number of requests which are searched in the J^{th} keyed file per unit time

For the Pointers File:

P(J) = total number of pointers in the J^{th} file

LP = length of the pointers in bytes

SP(J) = total number of accesses to the Pointers File associated with the J^{th} index file, per unit time.

For the Master File:

NMR = total number of Master Records

LMR = average length of the Master Records

Sm = total number of records retrieved in the master file per unit time

Also assumed given are all the characteristics of the device, including its timings and costs described in the previous analysis. With the previous information, the search time for each index file can be obtained using the program STOCAP and Equation (1). The desired factors will be designated by:

$K(J)$ = search time of the J^{th} index file when this is not partitioned and is organized as an Index Sequential File.

The following notations will be used for the other variables:

$n(J,h,I)$ = number of records in the I^{th} partition of the J^{th} index file, when this is partitioned into h groups ($I \leq h$)

$ST(J)$ = storage requirements for the J^{th} index file

$T(J)$ = total processing time of the J^{th} keyed file

$K(J,h,I)$ = search time for the I^{th} group of the J^{th} index file, when this is partitioned into h groups and each group is organized as an Index Sequential File.

An asterisk will be added to these values to designate their expressions when at optimality (minimum cost), e.g., $n^*(J,h,I)$.

$L_R(J,h,I)$ = record length of the I^{th} group of the J^{th} index file, when this is partitioned into h groups.

$Lk(J,h,I)$ = key length of the I^{th} group of the J^{th} index file, when this is partitioned into h groups.

These two last variables will be used for I<h only, because $Lk(J,h,h) = \max Lk(J)$ and $L_R(J,h,h) = \max L_R(J)$.

$Sk(J,h,I)$ = number of requests of the I^{th} partition of the J^{th} file when this is divided into h groups.

3.3.6.3.4 Procedure Description

Now, a detailed description of each step of the iteration will be presented.

Step 1: The storage requirements for each index file can be obtained by program STOCAP. Let this be ST_k.

The storage needed for the Pointer File can be expressed as the sum of all the number of pointers times the length of these pointers, or:

$$STp = LP\sum_J P(J) \qquad (21)$$

For the Master File:

$$STm = NMR \cdot LMR \tag{22}$$

The total storage, then, can be represented by:

$$ST = STk + STp + STm \tag{23}$$

If the cost of auxiliary storage is linear, then this cost can be expressed as:

$$STC = Cs \cdot ST \tag{24}$$

If not linear, then STC is evaluated using the corresponding function. The processing times can be obtained as follows:

For the Keyed Files:

$$Tk = \sum_{J} Sk(J)\ (Tk + T_B + K(J)) \tag{25}$$

And for the Unkeyed Files:

$$Tp + Tm = (TUNK + T_B) \sum_{J} Sp(J) + Sm(TUNK + T_B) \tag{26}$$

where SP(J) is the total number of requests for the existing pointer file or equivalently, the total number of successful searches in files organized under Structure 4.

Then, the Total Processing Time is given by:

$$TPT = Tk + Tp + Tm \tag{27}$$

and its cost can be obtained from Figure 15. Having the total cost and processing time, these can be plotted in the corresponding graph.

Step 2: The process for obtaining the group sizes when Assumption 8 holds, varies considerably compared to the case when it does not hold. Under Assumption 8, a good approximation of the optimal group sizes, when the file is divided into h groups, is obtained using the following expressions:

$$N^*(J,h) = \frac{1}{h}\ [K(h)]\ [D(J,h)] \tag{28}$$

where

$$N^*(J,h) = \begin{bmatrix} n^*(J,h,1) \\ n^*(J,h,2) \\ \vdots \\ n^*(J,h,h) \end{bmatrix} \tag{29}$$

and

$$D(J,h)=\begin{bmatrix} n(J) \\ Q(J)\ (T_k + T_B) \\ Q(J)\ K_2 \\ Q(J)\ K_3 \\ \vdots \\ Q(J)\ K_h \end{bmatrix} \tag{30}$$

$$\text{where } Q = \frac{C_T\ Sp(J)}{Cs(\max\ L_R(J) - \min\ L_R(J))} \tag{31}$$

NOTE: When index file J is of Structure 5a, i.e., without a separate Pointer File, then SP(J) is taken as the total number of successful searches in that file.

$K(h)$ is a matrix that can be obtained from Fig. 22.

FIG. 22:
MATRIX FOR K

$$K(2) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & -1 \end{bmatrix}$$

$$K(3) = \begin{bmatrix} 1 & 3 & 2 & 1 \\ 1 & 0 & -1 & 1 \\ 1 & -3 & -1 & -2 \end{bmatrix}$$

$$K(4) = \begin{bmatrix} 1 & 6 & 3 & 2 & 1 \\ 1 & 2 & -1 & 2 & 1 \\ 1 & -2 & -1 & -2 & 1 \\ 1 & -6 & -1 & -2 & -3 \end{bmatrix}$$

$$K(5) = \begin{bmatrix} 1 & 10 & 4 & 3 & 2 & 1 \\ 1 & 5 & -1 & 3 & 2 & 1 \\ 1 & 0 & -1 & -2 & 2 & 1 \\ 1 & -5 & -1 & -2 & -3 & 1 \\ 1 & -10 & -1 & -2 & -3 & -4 \end{bmatrix}$$

$$K(6) = \begin{bmatrix} 1 & 18 & 5 & 4 & 3 & 2 & 1 \\ 1 & 12 & -1 & 4 & 3 & 2 & 1 \\ 1 & 6 & -1 & -2 & 3 & 2 & 1 \\ 1 & -6 & -1 & -2 & -3 & 2 & 1 \\ 1 & -12 & -1 & -2 & -3 & -4 & 1 \\ 1 & -18 & -1 & -2 & -3 & -4 & -5 \end{bmatrix}$$

When Assumption 8 is not valid, then the optimal group sizes will be determined by Program OPT PART, which is designed to accept any given distribution of key lengths.

Selecting the Optimal Number of Groups. Due to the lack of convexity of the processing cost versus total processing time curves for each file and the total system, we cannot assume that the next best structure is found when each file is successively partitioned into a number of segmented groups, each structure being one group larger than the present structure. For this reason, at each iteration, we need to investigate the possibility of partitioning each group independently into h+1, h+2,...h+ΔH. groups, where h is the number of partitions in the present solution, and ΔH is a parameter that indicates the maximum extra number of partitions that will be considered for each file in every iteration. A suggested value of ΔH is 3, because when the number of partitions is more than 4 for a file, the changes in time and storage are expected to decrease and will not yield more significant changes in total cost than the partitions with a smaller number of groups.

Determining the Optimal Group Sizes. In order to determine the optimal group sizes, whether under Assumption 8 or not, the value of C_T (the slope of the cost of total processing time curve, evaluated at the total processing time corresponding to the present solution) is required. This value may be only a poor approximation for evaluating the optimal group sizes when the partitions are h + Δ H. For this reason, it will be better to re-evaluate C_T prior to every evaluation of the optimal group sizes. This can be achieved as follows for each file:

a. Evaluate the optimal group sizes for h + 1, using the value of C_T when the file was divided into h groups.

b. With the groups obtained, calculate the change in processing time from h to h + 1.

c. Get the corresponding C_T when the change calculated in (b) is added to the total processing time.

d. Evaluate the optimal group sizes for h + 2 using the value of C_T obtained in (c), etc.

Step 3: Having the optimal sizes for the partitions that will be considered for each file, the storage capacities and processing time can be obtained using program STOCAP and relation (a).

Before using program STOCAP, the record lengths and key lengths for each group have to be evaluated.

When Assumption 8 holds, these values are given by:

$$L_R(J,h,I) = \min L_R(J) + \left[\frac{\max L_R(J)-\min L_R(J)}{n(J)}\right] \sum_{k=1}^{k=I} n^*(J,h,I) \quad (32)$$

and

$$L_k(J,h,I) = \min L_k(J) + \left[\frac{\max L_k(J)-\min L_k(J)}{n(J)}\right] \sum_{k=1}^{k=I} n^*(J,h,I) \quad (33)$$

When Assumption 8 is not valid, then the key and record length can be obtained using their corresponding distributions and the $n^*(J,h,I)$ obtained in Step 2.

The Total Storage requirement for each partitioned index file is obtained by:

$$ST^*(J) = \sum_{I=1}^{I=J} ST^*(J,h,I) \quad (34)$$

where $ST^*(J,h,I)$ is the storage requirement for the optimal I^{th} group of the J^{th} file when this is partitioned into h groups.

The Total Storage requirement for the system is then given by:

$$TST = \sum_J ST^*(J) + STp + STm \quad (35)$$

where only one of the $ST^*(J)$ differ from the present structure.

The Total Processing Time for a segmented index file is given by:

$$T(J) = \sum_{k=1}^{k=h} [Sk(J) - \frac{Sp(J)}{n(J)} \sum_{I=1}^{I=k-1} n^*(J,h,I)] \, (T_k+T_B+K^*(J,h,k)) \quad (36)$$

Then, the Total Processing Time for the System becomes:

$$TPT = \sum_J T(J) + Tp + Tm \quad (37)$$

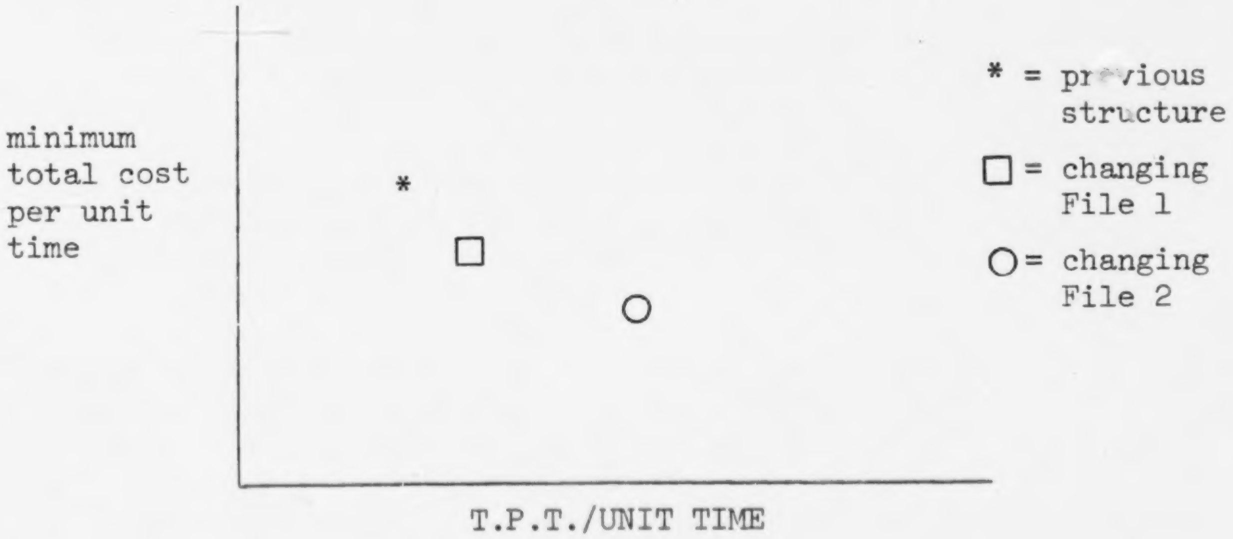
where Tp + Tm is the same as that obtained by expression (26).

Having the total processing time and storage values for each structure under consideration in the present iteration, then their corresponding costs can be obtained.

Step 4 - The proper selection of the change to be adopted cannot be selected on the criterion of Minimum Cost change. Some large files may yield a large reduction in cost, but at the same time a large change in Total Processing Time. This could skip some important point in the graph.

To illustrate the problem, suppose that the points corresponding to the previous structure and the ones corresponding to changes in different files are those shown in Fig. 23.

FIG. 23:
TOTAL COST/TOTAL PROCESSING TIME



It is clear that the change of File 1 yields an important point that is missed when the criterion of Minimum Cost change is used.

Consequently, the criterion of minimum slope has been adopted (min $\frac{\Delta C}{\Delta T}$). In this case, it is clear that the change of file 1 of the example will be selected.

Hence, for each of the new structures that are being considered in the iteration, we evaluate $\frac{\Delta C}{\Delta T}$ using the expression:

$$\Delta_{NS} = \frac{\Delta C}{\Delta T} = \frac{C_{NS} - C_{PS}}{T_{NS} - T_{PS}} \qquad (38)$$

where C_{NS} and T_{NS} have been evaluated by Step 4 and C_{PS} and T_{PS} are the total cost and processing time corresponding to the present solution and have been plotted already by Step 5 (or Step 1).

Steps 5 and 6 are self-explanatory and do not require any special formulas.

3.3.7 Segmentation of Index Files Based on Usage Frequency

Up to now, the file activity of the individual records has not been considered. This distribution was not necessary for the previous analyses because of Assumptions 6 and 7. Next we will explore the possibility of dividing an index file when a different usage frequency for the file occurs.

The reordering of records within an index file is not possible because the Index Sequential Organization uses the concept of High or Equal Key, which underlines the need of a lexicographic ordering. Moreover, even if reordering were possible, the search in an index file has to go through each index level (see Equation 1), and the time to go from one index file to another (T_B), which represents the more time consuming process, will always be present; therefore, no significant reduction in processing time can be achieved.

Another alternative, however, seems plausible for certain cases: a partition of an index file into two separate files, each organized under Index Sequential. One will contain the records of high usage frequency which will be searched first and the other, the records of low usage frequency.

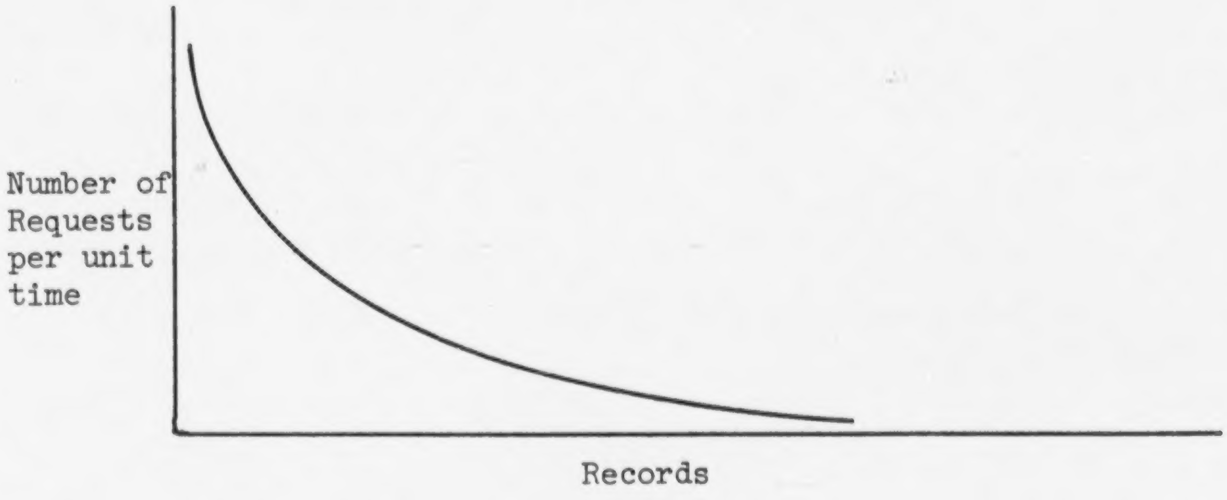
The total storage requirements will be assumed to remain essentially the same. This implies that the storage requirement for an index file varies linearly with the number of records.

Hence, we need only to compare the search times to determine whether this alternative is convenient or not.

(Comment: Program STOCAP was run for different number of records of constant key and data length, and the results confirmed this assumption).

For this analysis, we also assumed as given the usage frequency of the records of each index file. When the records are ordered in decreasing usage rate, we obtain a distribution as shown in Fig. 24.

FIG. 24:
USAGE FREQUENCY DISTRIBUTION



When the J^{th} index file is divided into two groups, then its processing time is given by:

$$T_2=S_k(J)\ [T_k+T_B+K(J,2,1)] + [S_K(J)-Sp(J,2,1)]\ [T_K+T_BK(J,2,2)] \tag{39}$$

where $Sp(J,h,I)$ = number of successful searches in the I^{th} group of the J^{th} index file, when this is divided into h groups. $K(J,2,1)$ and $K(J,2,2)$ are the average search times for the first and second group of the J^{th} index file when this is divided in 2 groups. These searched times are evaluated as before using Equation (1) and program STOCAP for a given partition.

The processing time for the J^{th} index file when this is not partitioned, is given by (2), which is equivalent to (40) with new notation:

$$T_1 = S_K(J)\ [T_K+T_B+K(J)] \tag{40}$$

It will be advantageous to partition when the following condition holds:

$$T_1>T_2 \qquad \text{or} \qquad T_1 - T_2>0 \tag{41}$$

After substituting (40) and (41) in (42) and rearranging, we get:

$$\frac{Sp(J,2,1)}{S_K(J)} > \frac{T_K+T_B+K(J,1)+K(J,2)-K(J)}{T_K+T_B+K(J,2)} \tag{42}$$

For any given usage distribution of the records, we can start increasing the size of the first group from 0 to n, where n is the total number of records, and test for condition (42) at each small increase. For each group size, the search times have to be reevaluated using Equation (1) and program STOCAP.

Note also that the distribution shown in Fig. 24 refers to the number of requests of the existing records; therefore, the area under the curve is equal to $Sp(J)$, and when this is divided into two groups, $Sp(J,2,1)$ is the area under the curve from 0 to the number of records in the first group.

If Condition (42) is satisfied for some partitions, the optimal size value of this partition is the one that yields the largest difference between the two sides of inequality (42).

This analysis can be performed independently from the other analyses discussed in this section. This is justified by Assumptions 6 and 7.

It may be convenient to perform first the analysis considering the index record length, i.e., the number of pointers (Structure 4 versus "mixed" structure); then the usage analysis just described; and lastly, the iterative procedure to select the index segments with respect to the key lengths.

Before each of these analyses is performed, each of the partitions that are organized as index sequential should be taken as an independent index file. For this, the proper $S_K(J)$, $S_K(J,h,I)$, Sp(J), and Sp(J,h,I) of each index file have to be carefully determined.

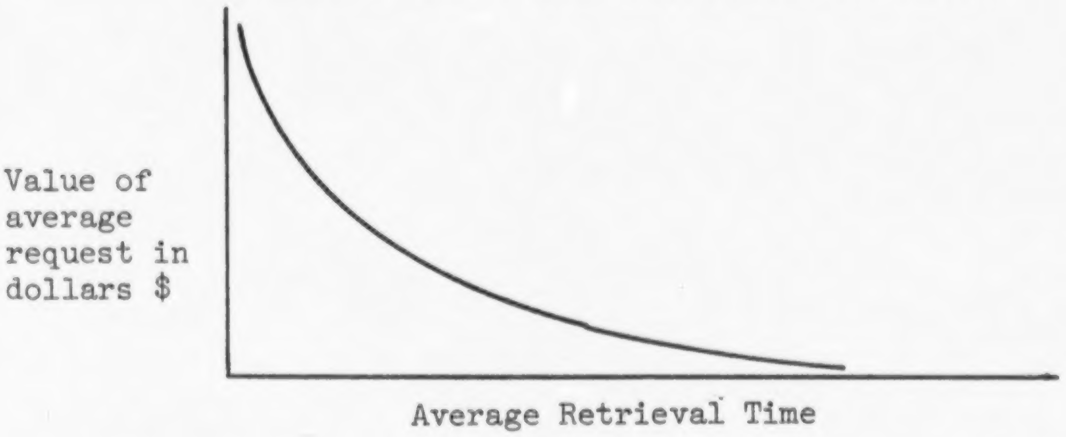
3.4 Consideration of Request Utility as a Function of Average Retrieval Time

It is generally agreed that the utility of file information decreases in some fashion as its retrieval time increases. This is probably true in the case of many users of the bibliographic type of system. And it is especially meaningful when there is available another type of storage system that could provide the information requested. In other words, as the time savings that are achieved using the computer system over a card catalog decrease, the value of former system decreases for the users.

It is possible, then, to obtain some kind of a curve that will represent the value of a request versus the length of its average retrieval time. Transforming this subjective value into some kind of cost equivalent, this curve can be related to the previous analysis.

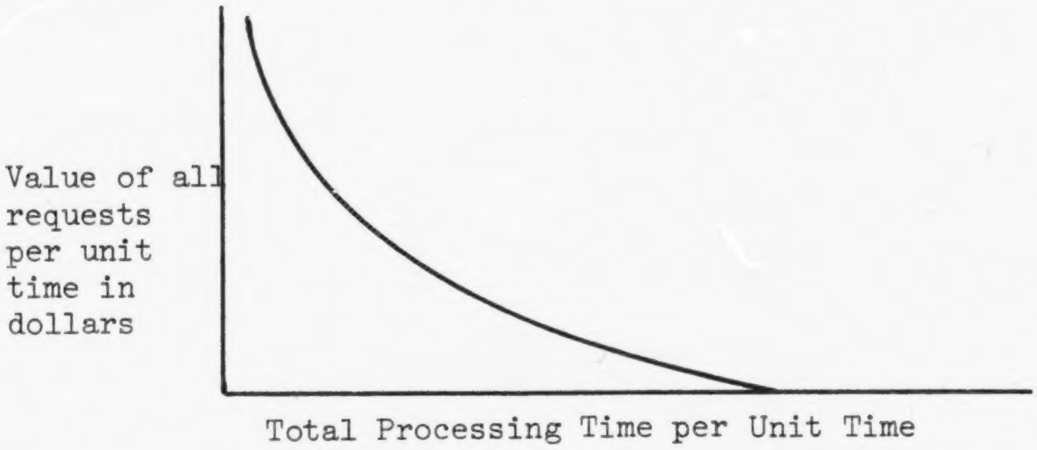
Suppose that the value of an average request versus its average retrieval time is expressed as in Fig. 25.

FIG. 25:
AVERAGE REQUEST $/AVERAGE RETRIEVAL TIME



Then, by multiplying each coordinate by the total number of requests per unit time, we obtain another curve as shown in Fig. 26.

FIG. 26:
ALL REQUESTS $/T.P.T.



The negative of these values can be interpreted as a cost assigned to each value of the processing time; therefore, we can absorb this curve in the previous analyses by subtracting the curve of Fig. 26 from the cost curve of Fig. 15, and using the resulting curve as the cost of total processing time.

In the previous analyses, there were no assumptions made on the actual shape of the cost curve of Fig. 15; therefore, no alterations are needed when this alternate cost curve is used.

APPENDIX

PARAMETER DEFINITIONS FOR FILE STRUCTURE ANALYSIS

Symbol	Definition
BI	Block Length (ISAM; with Overhead)
C	Number of Cylinders
CIT	Number of Tracks in Cylinder Index
C_{NS}	Total Search Cost of Next Solution
C_{PS}	Total Search Cost of Present Solution
C_s	Cost of Auxiliary Storage
C_T	Cost of Total Processing Time
h	Number of Groups into which Index File is Divided
IBN	Block Length of Last Block (With Overhead)
IBT	Blocks Per Track
IDL	Data Field Length (ISAM)
IEN	Logical Record Length of Last Index Record
IOTC	Number of Overflow Tracks Per Cylinder
IPTDC	Number of Prime Data Tracks Per Cylinder (ISAM)
IRB	Number of Logical Records Per Block
IT	Number of Track Index Entries Per Track
ITC	Track Capacity (Bytes)
ITPC	Number of Tracks Per Cylinder
IZ	Constant (Last Record, Keyed)
K	Average Search Time in ISAM File
K_L	Key Length
K_1	Average Search Time, Index File Segment ST_4
K_2	Average Search Time, Index File Segment ST_5

LBLK	Block Length (Data Only)
L_k	Corrected Key Length
LMR	Average Record Length, Master Records
LP	Record Length of Pointer Record (Bytes)
LR	Total Index Record Length (Includes all levels)
Lu	Actual Minimum Key Length for Uniqueness
m	Number of Master Records
M	See "RM", below
n	Total Number of Discrete Key Values
NB	Number of Blocks Per Track
NIL	Number of Index Levels (ISAM)
N(J)	Number of Records in Index File Segment J
N(M)	Number of Keys Associated with M Master Records (Largest Group)
NT	Total Number of Tracks, Structure 4 Index File
NTM	Minimum Number Tracks at Given Level That Will Justify a Higher Index Level
NT1	Total Number of Tracks Needed for Structure 4 Segment of Index File Organized in "Mixed" Form
NT2	Total Number of Tracks Needed for Structure 5 Segment of Index File Organized in "Mixed" Form
PDT	Number of Prime Data Tracks
P(J)	Total Number of Pointers in the J^{th} File
RM1	Number of Tracks in First Master Index (ISAM)
RM2	Number of Tracks in Second Master Index (ISAM)
RM3	Number of Tracks in Third Master Index (ISAM)
RNT	Total Storage Required (Tracks)
ROT	Time Required to Read 1 Track (One Full Rotation in a Disk DASD)
RT	Records Per Track
Sk	Total Number of Requests Per Unit Time (Number of Key Searches)
Sm	Total Number of Records Retrieved (Number of Accesses to the Master File)
Sp	Total Number of Successful Searches (Number of Accesses to Pointers File under Structure 4)

SP(J)	Total Number of Accesses to Pointers File Associated with Jth Index File, Per Unit of Time
STC	Total Cost of Auxiliary Storage
STk	Storage Required for Index File
STp	Storage Required for Pointers File
TB	Seek Time
TI	Number of Tracks in Track Index
Tk	Non-DASD Timing Factors, Keyed File (Constant)
T_m	Non-DASD Timing Factors, Master File
T_{NS}	Total Search Processing Time of Next Solution
T_p	Non-DASD Timing Factors, Pointers File
T_{PS}	Total Search Processing Time of Present Solution
TPT	Total Search Processing Time
Tunk	Non-DASD Timing Factors, Non-Keyed File (Constant)
X	Device Constant
Y	Device Constant

4. THE ANALYSIS OF BIBLIOGRAPHIC FILE STRUCTURES, USING INDEXED SEQUENTIAL ORGANIZATION

By Jorge Hinojosa

4.1 Introduction

Different file structures can be used in an information retrieval system. The choice of a specific type of hardware and the operational goals of the system determine the feasibility or unfeasibility of such structures.

The present study is concerned with an information retrieval system for bibliographic catalogs using direct access storage devices. The system is formed of different index sequential files organized under some feasible structure which may be the same for all files. The index files provide access to a master file composed of MARC-structure records.

A measure of retrieval performance of the system would have two main properties. First, it would express the ability of the system to accomplish the goals for which it was created, or its "effectiveness." Second, it would consider factors related to cost, or "efficiency." We are concerned with the second property.

4.1.1 Object

The objective of the project is to maximize the efficiency of the system, or what is equivalent, to minimize its operational cost. The specific criterion is to obtain a minimum cost per request put to the system by a person searching the computer data base.

4.1.2 Cost Relations

Two main costs are considered:

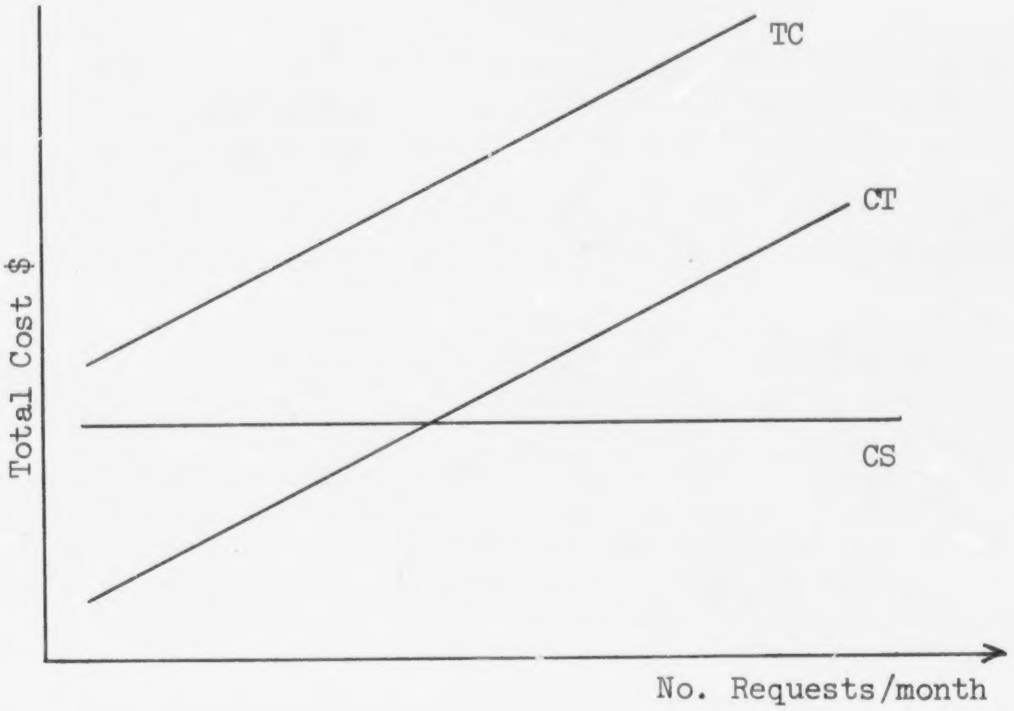
a. Cost of Auxiliary Storage (Cs). This is the cost of the Direct Access Storage Device (DASD) space used and is assumed to vary linearly with the amount of information stored.

b. Cost of Processing Time (Ct). This represents the cost of using the central processing unit (CPU). In general, this cost is calculated as follows:

COST = Rate (CPU+ALPHA·PP) + F(CR, LP, CP) where
CPU = Central processing time (minutes)
ALPHA = A function of central memory used
PP = Peripheral processor time (minutes)
F (CR, LP, CP) = A function of the number of input cards, lines printed and cards punched.

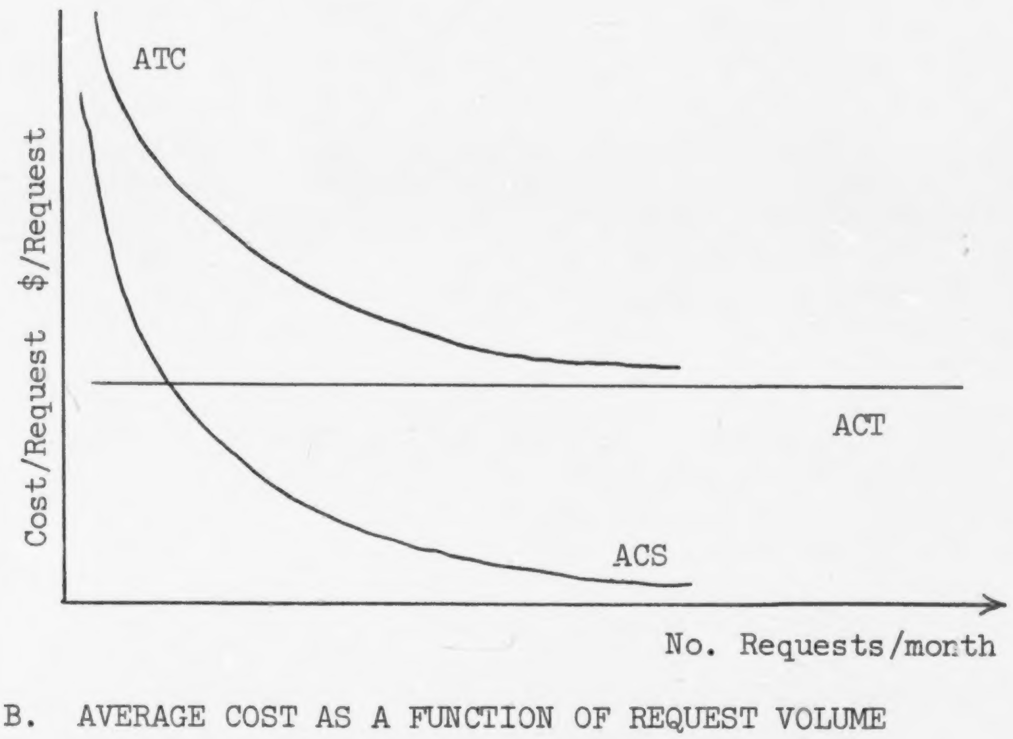
We are not concerned with F(CR,LP,CP), and we assume that the amount of central memory used and peripheral processor time are not affected by changes in the file structure. We also assume that the central processing unit is used all the time during the search and retrieval stage of system operation. Under these assumptions, the cost of total processing time varies linearly with respect to retrieval time.

FIG. 27:
OPERATIONAL COST OF A FILE



A. TOTAL COST AS A FUNCTION OF REQUEST VOLUME

TC = total cost
CT = cost of time
CS = cost of storage



B. AVERAGE COST AS A FUNCTION OF REQUEST VOLUME

ATC = average total cost
ACT = average cost of time
ACS = average cost of storage

If the system consists of J files, the total operational cost of the system for a given period of time is given by:

$$TCS = C_s \sum_{j=1} S_j + C_t \sum_{j=1} T_j$$

For each of the files, both storage (S) and time (T) are a function of the file structure, and so is the operational cost (C) of the file. Hence, TCS is better expressed as:

$$TCS = \sum_{j=1}^{J} TC_j = \sum_{j=1}^{J} Q(E_j) \qquad (1)$$

where TC_j = Operational cost of file j

$TC_j = Q(E_j)$

E_j = Structure of file j

Because of the assumptions that cost per byte of storage for a given device and cost per minute of processing time are constant, it is clear that minimum operational cost of the system is equal to the sum of the minimum operational cost of the individual files.

$$\text{Min } TCS = \sum_{j=1}^{J\rightarrow} \text{Min } Q(E_j) \qquad (2)$$

It is important to emphasize that TCS is total cost per period of time and NOT total cost per request. This is because it is unlikely that all the files would be used with the same intensity. Furthermore, the optimal structure E_j that yields Min TC_j is conditioned to the expected usage frequency of the file j.

Fig. 27, A and B, illustrates the behavior of cost for a given file. Each different structure defines a set of curves.

4.1.3 The Problem

Given the expected usage frequency per period of time, the file size, and the data characteristics, find an optimal structure that yields minimum operational cost for the file, or equivalently, minimum cost per request.

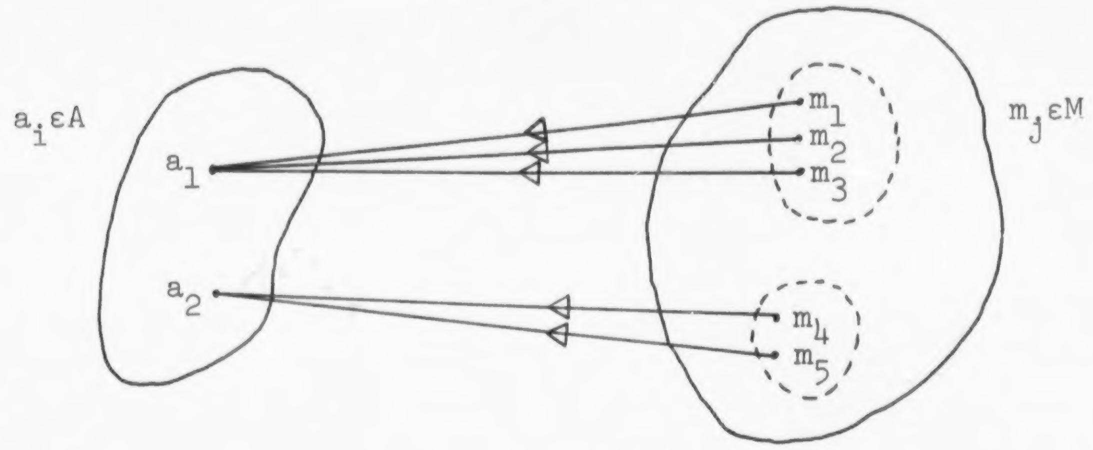
4.2 General System Structure

4.2.1 File Structures Selected for Analysis

4.2.1.1 Access Record Mapping

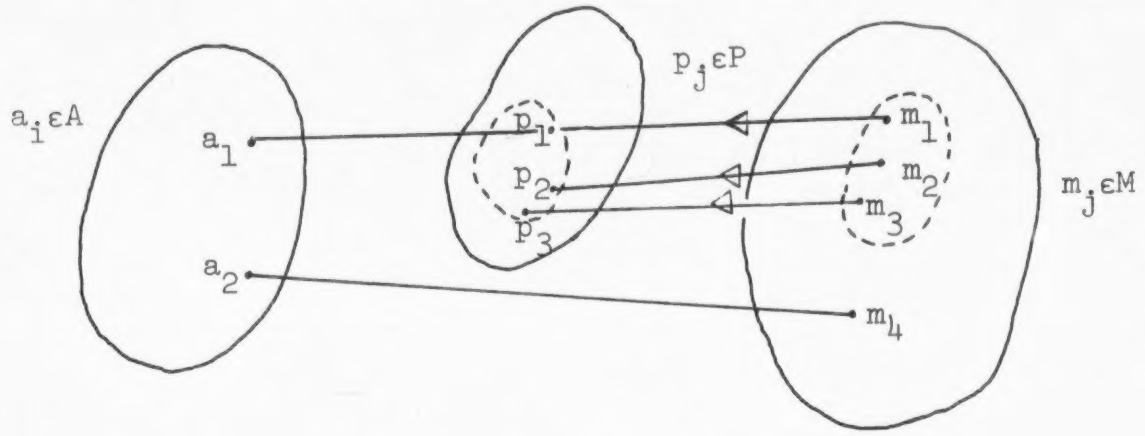
Let M = $\{m_j\}$ be the set of bibliographic records. (This set constitutes the Master File.) Let $A = \{a_i\}$ be the set of inverted access records that comprise an index file. There may be several index files in the system. If the subset of master records $M_i = \{mj\}$ $j=1,\ldots,k$ are indexed under the same index record a_i, we say that $M_j \varepsilon M_i$ maps into a_i. This mapping, shown in Fig. 28A, constitutes one of the feasible structures, say I. The elements of M_i mapping into A_i are not contiguous in M in a physical sense and might just as well map into an element of another access file. Hence, a_i needs to contain all addresses (pointers) pointing to $m_j \varepsilon M_i$.

FIG. 28A:
MAPPING OF MASTER RECORDS INTO ACCESS RECORDS FOR STRUCTURE I



If we define a set of addresses (or pointers) $P = \{pj\}$, such that there is a one-to-one correspondence of mj to pj ($mj \rightarrow pj$) and the $p_j \varepsilon P_i$; $P_i = \{pj;\ mj \longrightarrow pj, \forall mj \varepsilon M_i\}$ are contiguous in P in a physical sense, then a_i needs to contain only the address of the first $pj \varepsilon Pi$. If only one mj maps into a given a_i, we assume a direct mapping $mj \longrightarrow a_i$. This combination constitutes structure II and is presented in Fig. 28B.

FIG. 28B:
MAPPING FOR STRUCTURE II



4.2.1.2 Connection with Previous Analysis

To provide continuity with a previous analysis, the reader may note that Structure I in the present paper corresponds in part to Structure 5a in the prior chapter by Rodriguez and Structure II corresponds in part to Rodriguez' Structure 7. In that analysis, division of an index file into segments was proposed using either index record total length or key length as a basis. The diagrams for the previously defined Structure 5a and Structure 7 follow (Fig. 29). (The hachured areas are waste space in index record.)

FIG. 29A:
FEASIBLE STRUCTURES

STRUCTURE 5a - FIXED-BLOCKED UNIT LISTS

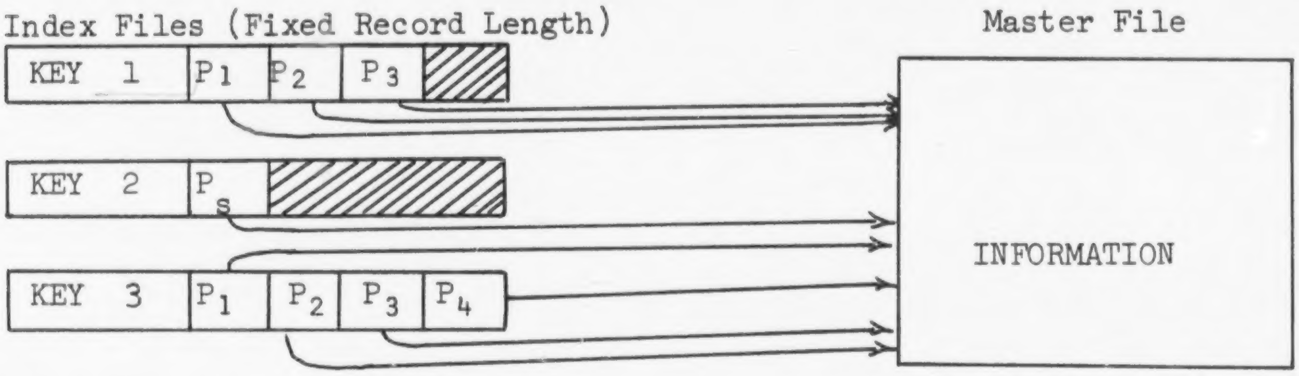
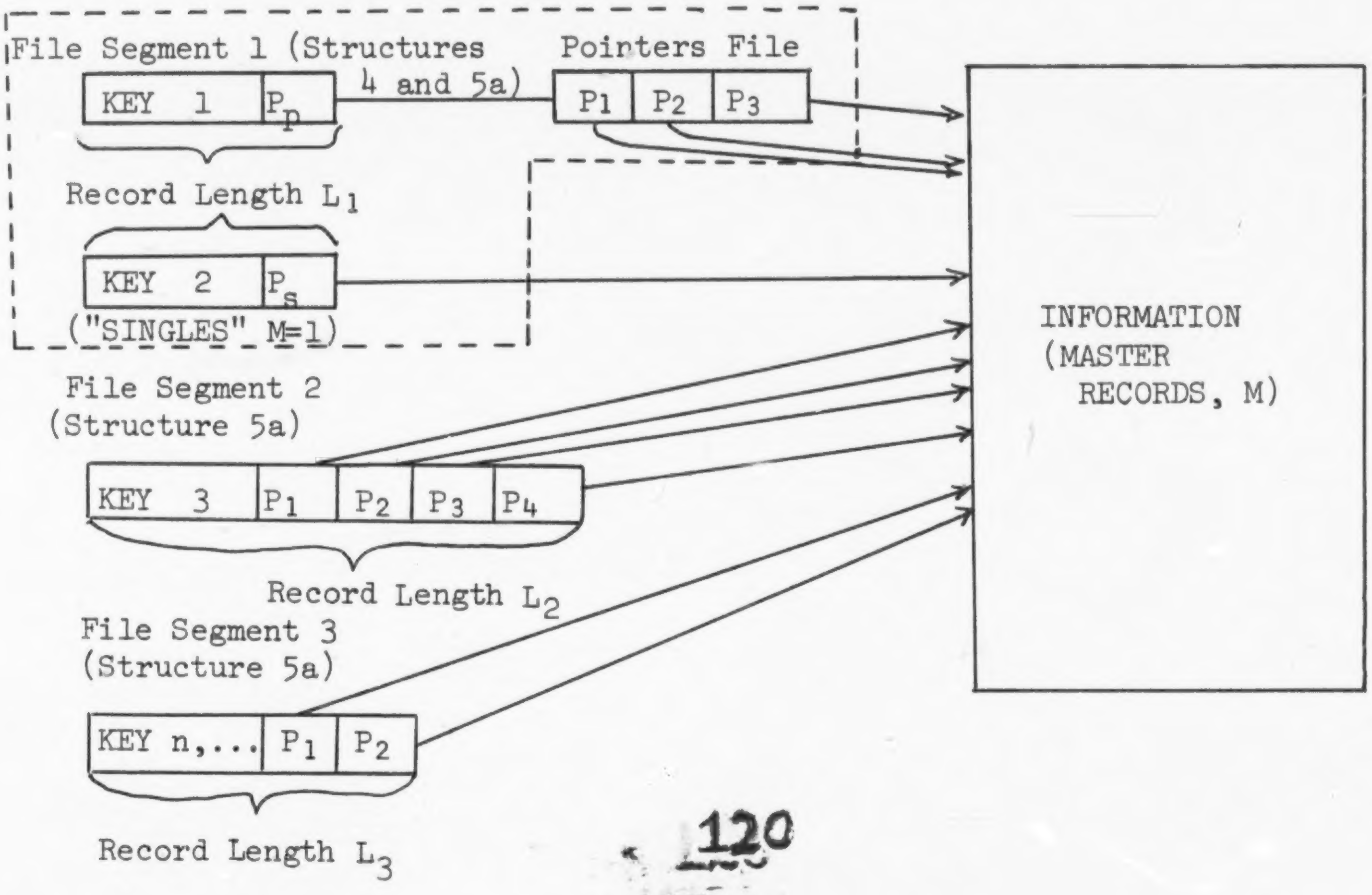

Index Files (Fixed Record Length)
Master File
KEY 1
P_1
P_2
P_3
KEY 2
P_s
INFORMATION
KEY 3
P_1
P_2
P_3
P_4

FIG. 29B:
STRUCTURE 7 - "MODIFIED MIXED"
Illustrating Segmentation Based on Index Record Length


File Segment 1 (Structures 4 and 5a)
Pointers File
KEY 1
P_p
P_1
P_2
P_3
Record Length L_1
KEY 2
P_s
("SINGLES" M=1)
File Segment 2
(Structure 5a)
INFORMATION
(MASTER
RECORDS, M)
KEY 3
P_1
P_2
P_3
P_4
Record Length L_2
File Segment 3
(Structure 5a)
KEY n,...
P_1
P_2
Record Length L_3

Note that Structure II in Fig. 28B is identical to the structure in File Segment 1 of Structure 7 (Fig. 29B) prior to division into a second, etc., file segment based on index record length, which was the basis for the original study. To avoid confusion, we shall continue to use the previously established structure designations.

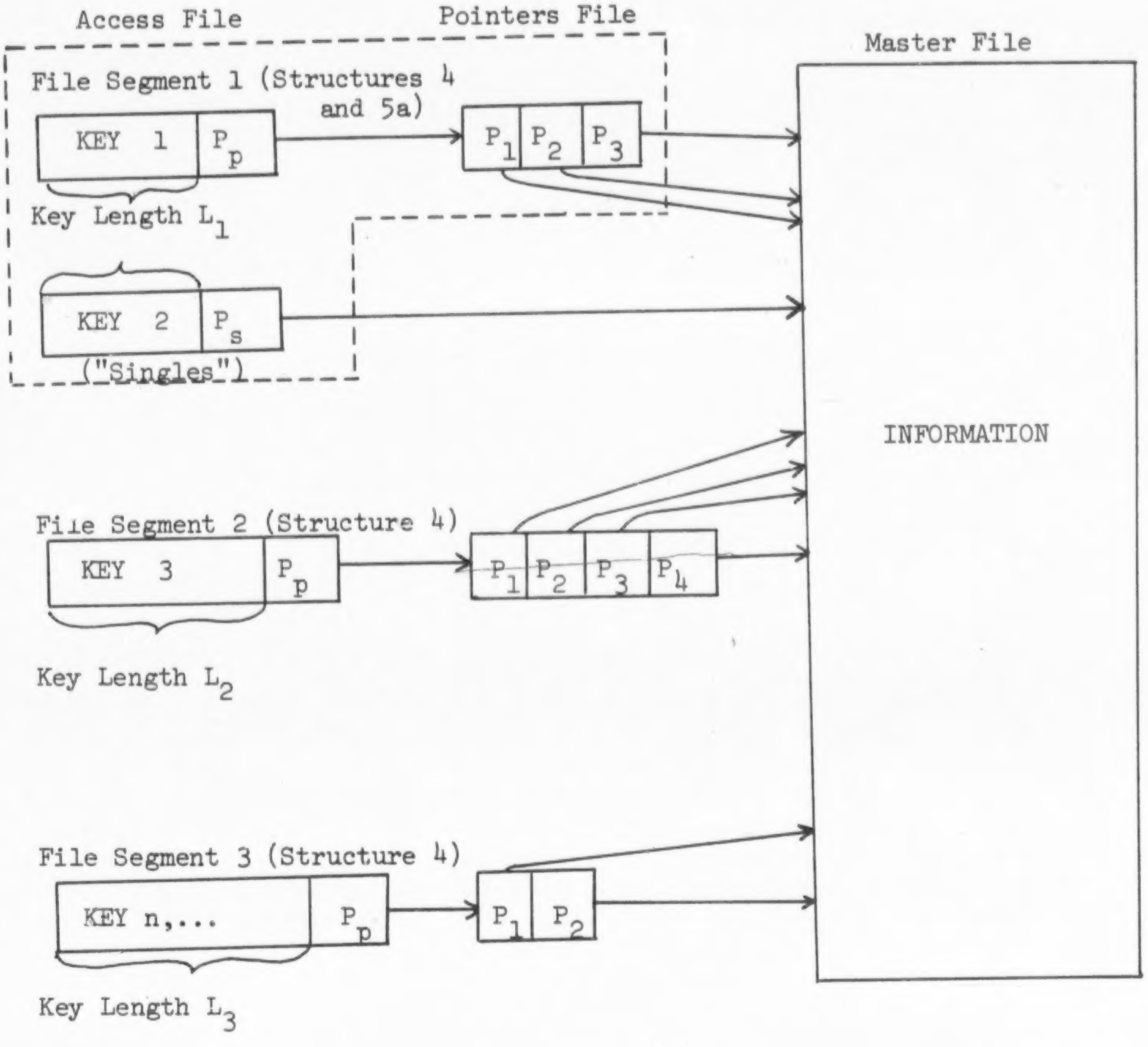
It is also important to note that the access records (index file) in Structures 5a and 7 must be of fixed length, and must be pre-set at file load time. In Structure 5a, this imposes a space penalty: record length must be set to that of the longest record. That is, the maximum record length corresponds to the key field length plus the number of pointers corresponding to the key a_i associated with the largest number of master records M_i. The existence of a dispersed, asymmetric distribution of key-lengths implies a consequent inefficient use of storage.

As an example of segmentation by some criterion using index record length, suppose the largest group of keys N(M) was associated with four master records, that is M=4. This group is extracted from File Segment 1, "Non-Singles", which is the initial file, and placed in File Segment 2 using Structure 5. (See Fig. 29B.) The process is then repeated recursively, as the model prescribes.

A variant of Structure 7 which provides the maximum flexibility, since it frees the analysis from need to attend to the ratio of the number of keys N associated with m master records, employs the distribution of key lengths solely as the basis of division. This structure is illustrated in Fig.30. It is this version of Structure 7 which supplies the basis of the remainder of this paper, since it is the form of access file that can be tested under software currently available to the ILR facility. Moreover, it permits future experimentation with optimal indexed sequential files using variable-blocked index record support now becoming available (IBM OS Release 18). At that time, it will be possible to eliminate the Pointers File (Addresses list).

FIG. 30:
STRUCTURE 7 - "MODIFIED MIXED" WITH TWO-LEVEL LISTS

Illustrating Segmentation Based on Key Field Length



Access File
Pointers File
Master File
File Segment 1 (Structures 4 and 5a)
KEY 1
P_p
P_1 P_2 P_3
Key Length L_1
KEY 2
P_s
("Singles")
INFORMATION
File Segment 2 (Structure 4)
KEY 3
P_p
P_1 P_2 P_3 P_4
Key Length L_2
File Segment 3 (Structure 4)
KEY n,...
P_p
P_1 P_2
Key Length L_3

4.2.1.3 Record Format

We shall assume that all index records are blocked. For keyed files, both key areas and data areas are fixed length; that is, the index record is fixed length.

4.2.2 Comparison of Structures 5 and 7

There are two factors that determine storage efficiency for index files: 1) distribution of key lengths (Fig. 31); and 2) master records mapping into an access record (Fig. 32).

FIG. 31:
KEY LENGTH DISTRIBUTION

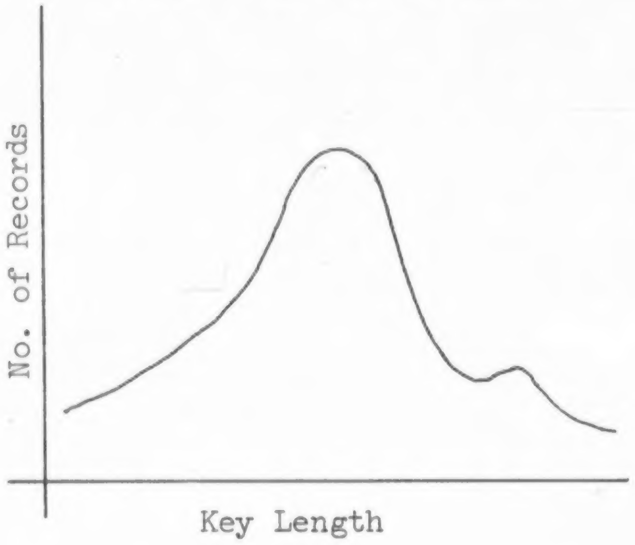
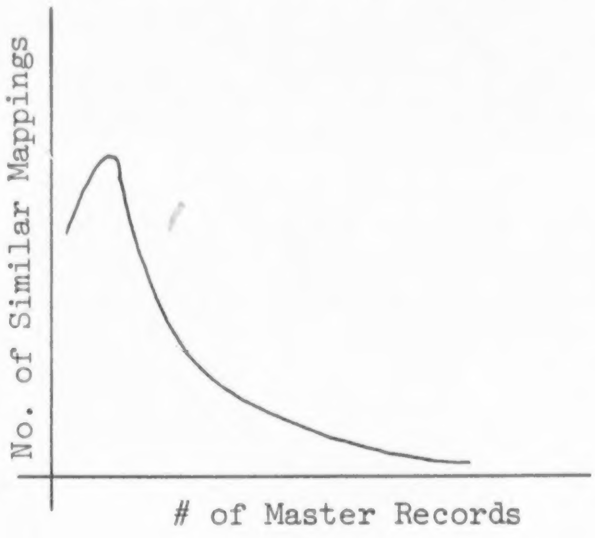


FIG. 32:
POINTERS DISTRIBUTION



Because of the restrictions on Indexed Sequential keyed records, it is clear that for Structure 5 there is a waste of storage in both key area and data area. For Structure 7, it occurs only in the key area (Fig. 29B and 30). Using Structure 5, all the information needed to retrieve the subset of related master records is reached with one search. Using Structure 7, we need an additional seek if the Pointers File has to be accessed. There are many other considerations to be taken into account before deciding which structure is better. Therefore, each one should be analyzed individually to determine how it can be improved.

Since the analysis by Rodriguez (Chapter 3) concluded that Structure 5, used alone, was inefficient for segmentation by record length due to the much larger required processing time (to transfer the search from segment to segment), we will concentrate our analysis on the "Mixed Structure," Structure 7, to see how it can be improved. Later analyses will be directed at comparing the two structures for suitability to a given file.*

4.3 Division of an Index File Based on Key Lengths

One way to avoid the waste of storage in the key area of the access file is to divide or segment the file into several groups based on key lengths. Records with short keys might be in one group, medium-length keys in another, and so on. However, the search time for some records would increase because we may need to look for those records in each of several files. Hence, a division would be feasible only if the total search cost per request for the divided file is less than the total cost per request for the undivided file.

The operational cost of a file using Structure 7 may be improved by a better utilization of the storage, minimization of search time, and by more efficient allocation of both overflow records and non-keyed (pointer) records. In what follows, we present different schemes for analyzing and improving the index and pointer files. The following assumptions are used in the analysis.

a. A discrete distribution of key lengths for the file under consideration is available.

b. The usage frequency for a group of records with a given key length is independent of the key length.

c. The usage frequency for a group of records is proportional to the number of records in the group. This assumption can be interpreted in two ways: a) each record is equally likely to be requested, b) each group of records has the same usage frequency distribution. Obviously, the second interpretation is stronger.

d. Search is done sequentially,** beginning with the access group or file segment having the largest number of index records. This follows from assumption 3.

---

[*] The index record must be of fixed length in the currently available version of the manufacturer-supported Indexed Sequential Organization. A future release promises to support variable-length records (although key field must still be fixed length). This means that when implemented, Structure 5 would probably supplant Structure 7 in efficiency, because the need for a separate Pointers File would be obviated. The efficiency of the software implementation is, of course, another issue for testing to resolve.

[**] That is, groups placed in files f_1, f_2, f_3..., f_n must be searched successively to respond to a search request, particularly in light of the part-key (i.e., key prefix) capability of the Indexed Sequential Access Method. For this reason, attention must be given to the need for both explicit stopping signals prior to the last file segment, and to algorithmic stopping rules for multi-group searches, to minimize retrieval time.

e. If the file requires master index levels, the highest of these is stored in core, if possible.

f. The cost per unit of storage and cost per unit of time are constant for a given period of time.

4.3.1 Cost of Storage

Let F denote the undivided index file, containing a set K of records with key lengths k_j which have a density function $g(k_j)$ and a distribution function $G(k_j)$. If we divide F in n subfiles or segments, F_i, the index i is increasing as the key length increases.

The storage requirement ST(K) of file F is proportional to the longest key (kx) and the number of records in the index file.

$ST(K) = C \cdot k_x \cdot G(k_x)$, where C is a proportionality factor that takes into account the different index levels. We assume that C is constant for all subfiles. Then,

$$ST(n,K,D_i) = \sum_{i=1}^{n} ST(K_i^{!}) = C \sum_{i=1}^{n} k_i[G(k_i) - G(k_{i-1})] \tag{3}$$

Where $Di = \{k_i\}$ is the set of division points such that $Di>Dj$ if $k_1^i>k_1^j$.

We shall prove that for a given n, the total storage and, therefore, the storage cost of a file F divided in n subfiles is a unimodal function, and its central part behaves as a convex function of the set of division points Di.

Let's assume that n = 2, and k_i, k_j are the corresponding division points for two different sets D_i and D_j, such that $k_j>k_i$. We have that,

$$ST(2,K,D_i) = Ck_iG(k_i) + Ck_x\,[G(k_x) - G(k_i)] \tag{4}$$

$$ST(2,K,D_j) = Ck_jG(k_j) + Ck_x\,[G(k_x) - G(k_j)] \tag{5}$$

Then, if $k_l = \lambda k_i+(1-\lambda)k_j$ and $0\leq\lambda\leq1$, the function ST(2,K,D) is convex if:

$$\lambda ST(2,K,D_l) + (1-\lambda)ST(2,K,D_j) > ST(2,K,D_l) \tag{6}$$

Using (4) and (5) we reach the conclusion that (6) is true if:

$$\lambda> \left(\frac{G(k_j) - G(k_l)}{G(k_j) - G(k_i)}\right) \left(\frac{k_x-k_l}{k_x-k_i}\right) \quad \text{holds.} \tag{7}$$

It is obvious that for the central values of the distribution function G(x), the first parenthesis in (7) is almost equal to λ, and the second parenthesis is always less than one. Therefore, (7) holds. This implies that there exists only one set of division points for which the storage cost of the divided file is minimum.

Subtracting (5) from (4) and eliminating common terms, we obtain the expression,

$$ST(D_i) - ST(D_j) = G(k_j)\ (k_x - k_j) - G(k_i)(k_x - k_i) \qquad (8)$$

We can see that for small values of k_j and k_i, the expression above is positive and for very large values of k_j and k_i, (8) is negative, which is the property of the unimodal functions.

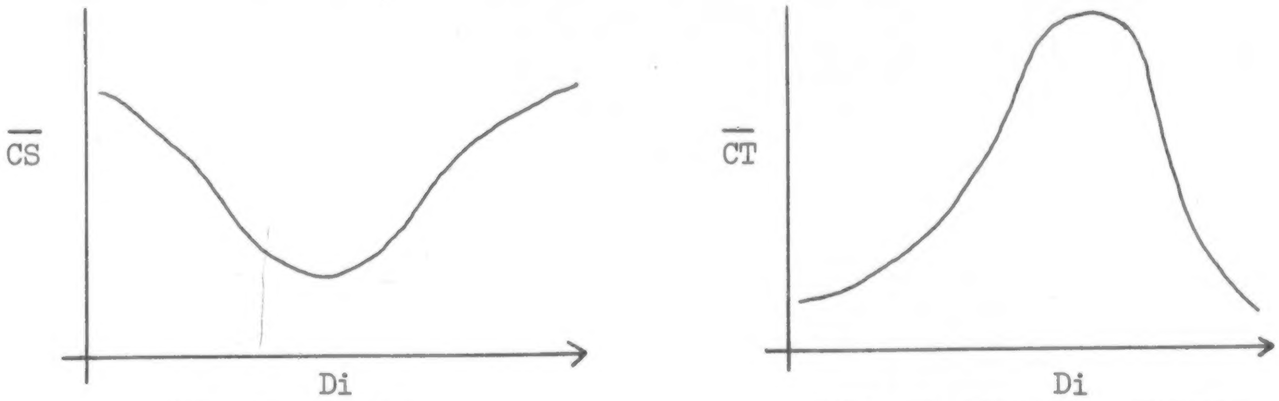
4.3.2 Cost of Time

By assumptions (3) and (4), search time is also a function of the set of division points Di, and its value is increasing as the number of records per subset becomes equal. Hence, the shape of the search-time cost as a function of division points Di is a positive unimodal function.

Furthermore, we notice that for a given structure, the cost of storage per request is proportional to 1/NR* while the cost of search time per request is constant. We conclude that a minimization of the combined cost per request cannot be achieved independently of request volume. However, we can find an optimal set of division points Di that minimizes $\overline{TC} = \overline{CS} + \overline{CT}$ for a given File size-Request Volume relation α.

In Fig. 32, TC denotes total cost, CS denotes cost of storage, and CT, the cost of time. The bar above denotes average cost per request. $\alpha = NR/N$, where NR is the expected number of requests, and N is the total number of records in the index file.

FIG. 33:
COSTS FOR A DIVIDED FILE AS A FUNCTION OF THE
SET OF DIVISION POINTS (Di)



COST OF STORAGE

COST OF TIME PER REQUEST

---

*NR = number of requests.

From Fig. 33 it is possible to see that the combined cost function TC is not a "well behaved" function and that it might contain local minima. Therefore, to find the global minimum and be sure of its uniqueness, we have to consider the set of all possible division points Di, at which the first division occurs. For a set (F,G(k), n) in which the key-length distribution G(k) has a maximum number of points NK, we have to consider all the different combinations. The number of combinations can be expressed as:

$$\frac{NK!}{n!(NK-n)!}$$

4.3.3 Dynamic Programming Formulation

We state the problem as follows: given a file F, containing a set K of records with key lengths distributed as G(kj), find the set of (n-1) division points (n=2,3,...) that minimizes the total search cost per request TC(n,K). The total storage requirements of the file divided into n subfiles is,

$$ST(n,K) = \sum_{i=1}^{n} st(i,K'_i) \tag{9}$$

where $st(i,K'_i)$ is the storage requirement for the i^{th} subfile containing a subset Ki of records.

The combined search time per request under assumptions 3 and 4 is given by:

$$T(n,K) = t_1 + \sum_{h=2}^{n} \left(1 - \frac{1}{N} \sum_{j=1}^{h-1} N(j)\right) t_h \tag{10}$$

where N(j) is the size of the j^{th} subfile and t_h is the search time per request for the h^{th} subfile. The subscript i in equation (9) makes reference to position of the subfile within the key length distribution G(k). The subscripts j and h, make reference to the size of the file.

The total search cost per request is given by:

$$\overline{TC}\,(n,K) = ST(n,K)\cdot CSTR/NR + T(n,K)\cdot CTM \tag{11}$$

where CSTR = cost of storage in dollars per track,
CTM = cost of computer time in dollars per millisecond.

The principle of optimality of Dynamic Programming states that "an optimal policy has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

In applying this principle, let f(2,K') be the minimum total cost per request obtained by dividing a file containing a subset K' of records into TWO SUBFILES,

then

$$f(2,K') = \underset{\substack{(k_1 \le k_i < k_x) \\ 0<K' \le K}}{\mathrm{Min}} \{C[G_i k_i + (G_x - G_i)k_x]/NR + t_1 + (1 - N1/N)t_2\} \tag{12}$$

where t_1 = search time for the first subfile

t_2 = search time for the second subfile.

Let's define the first subfile as the subset of records K_1' containing all records with keys greater than or equal to the first key and smaller than or equal to k_i. Once we have found the key length k_i that yields $f(2,k)$ we can add the storage requirements of both subfiles and store it as one value which we call $S(2,K')$. The same optimization is carried out for all possible subfile sizes K'.

The minimum total cost for a file divided in THREE SUBFILES is obtained in a similar way.

$$f(3,K') = \underset{\substack{(k_1 \le k_i \le k_x) \\ 0 \le K' \le K}}{\mathrm{Min}} \left\{[CG_i k_i + S(2,K - K_i')]/NR + t_1 + \sum_{h=2}^{3} (1 - \frac{1}{N} \sum_{j=1}^{h-1} Nj)t_h\right\} \tag{13}$$

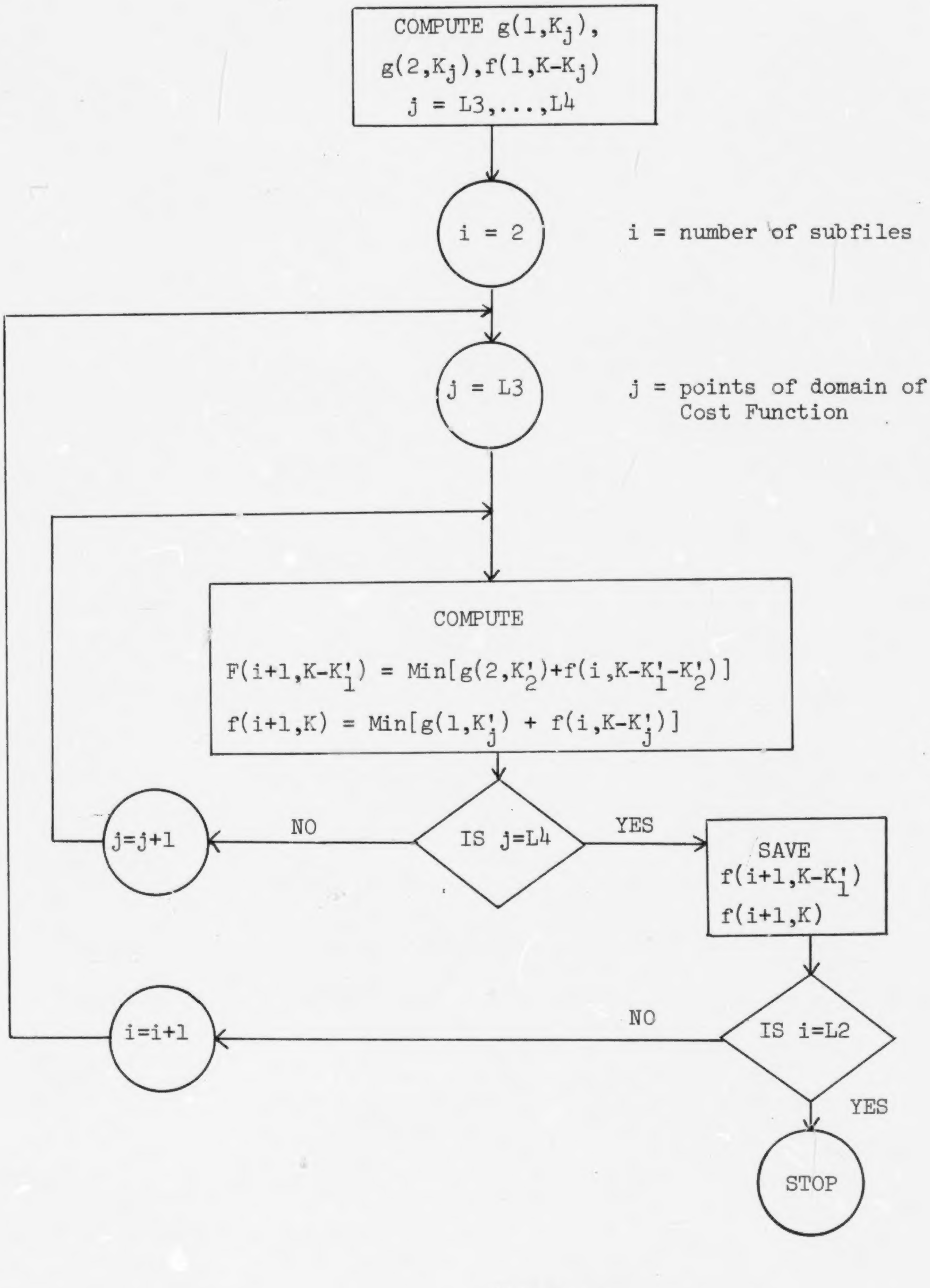
And in general

$$f(n,K') = \mathrm{Min} \left\{[CG_i k_i + S(n - 1, K - K_i')]/NR + t_1 + \sum_{h=2}^{n} (1 - \frac{1}{N} \sum_{j=1}^{h-1} N_j)t_h\right\} \tag{14}$$

It should be noticed that the search-time cost cannot be stored as one value because the subfiles' size is continuously changing. However, defining $g(1,K_i)$ as the cost apportioned to the first subfile containing a subset K_1' of records, it is possible to obtain a more general recursive relation, namely:

$$f(n,K') = \underset{\substack{0 \le K_i' \le K' \\ 0 \le K' \le K}}{\mathrm{Min}} \left\{g(1,K_i') + f(n-1, K' - K_i')\right\} \tag{15}$$

FIG. 34:
FLOW CHART FOR DYNAMIC PROGRAMMING COMPUTATIONS


COMPUTE $g(1,K_j)$, $g(2,K_j)$, $f(1,K-K_j)$ $j = L3,\ldots,L4$
$i = 2$
i = number of subfiles
$j = L3$
j = points of domain of Cost Function
COMPUTE
$F(i+1,K-K'_1) = \mathrm{Min}[g(2,K'_2)+f(i,K-K'_1-K'_2)]$
$f(i+1,K) = \mathrm{Min}[g(1,K'_j) + f(i,K-K'_j)]$
$j=j+1$
NO
IS $j=L4$
YES
SAVE $f(i+1,K-K'_1)$ $f(i+1,K)$
$i=i+1$
NO
IS $i=L2$
YES
STOP

4.3.4 Solution of the Problem

The recursive relation presented above suggests the use of a general numerical procedure similar to the one described in the flowchart of Fig. 34. However, there exist some difficulties due to the use of assumptions 3 and 4 in the computation of search time for file F, when divided into *n* subfiles. The optimal values $f(i, K-K_j')$ cannot be stored as cost, but rather as the combined storage requirement for the group of subfiles F'i in $(K-K_j)$, and the individual search times for each F'i. This is because the size of each subfile is changing, and we need to know at each step of the optimal value computation $[f(i+1,K) = g(1,K_i)+f(i,K-K_i')]$ what is the ordering according to size of the (i+1) subfiles.

Therefore, as *i* increases, the number of values (which we have to store and later compare) also increases, making it difficult to use the same loop to evaluate the optimal values for different numbers of divisions.*

4.3.5 Sample Computations and Results

Using a key-length distribution obtained from "Author File" and the characteristics of the IBM 2314 DASD (as given in Fig. 36), we divided a file in h = 2,3,4,5 subfiles for eight different file size/request volume relations. (Division was recursive.)

FIG. 35:
PARAMETERS AND COST VALUES USED

PARAMETER	VALUE
T_B: Access Motion Time	75.0 msec.
T_K: Data Transfer Time	25.0 msec.
ROT: Rotational Delay Time	25.0 msec.
Unit of Time	A month
Cost of Computer Time	$80.00/hr. (IBM 360 Model 40,108K Core Partition)
Cost of Auxiliary Storage	$.00002142/byte-month, or .15625/ track-month (2314 DASD)

---

*In our subroutine DYNPART, coded to test the proposed solution, we do not use the outmost loop shown in the flow chart. Storage requirements are calculated using subroutine STOCAP; search times are calculated using subroutine TIEMPO. To appreciate the computational power of DYNPART, we compare the number of calculations and computer time. When using direct enumeration methods, we needed to call subroutines STOCAP and TIEMPO approximately five million times, to divide a file in 1, 2, 3, 4, and 5 parts. The average computational time was 1.5 minutes. DYNPART uses subroutines STOCAP and TIEMPO only 1,200 times for doing the same job and takes an average time of three seconds.

The following results were obtained:

Table 6: Min. Total Search Cost per Request (Dollars)

NR/N	NUMBER OF SUBFILES (SEGMENTS)				
	(No Division) 1	2	3	4	5
.01	.200135	.14276	.13074	.12776	.12554
.02	.10245	.07408	.06914	.06794	.06762
.05	.04384	.03288	.03203	.03184	.031768
.10	.024307	.01915	.01894	.018908	
.20	.01454	.01222	.01217		
.50	.008677	.007989	.00797		
.70	.007561	.007147	.00714		
1.00	.006724	.006509	.006505		

From the results presented above, we deduce an effect of increase in volume of searches, viz., that as the relation NR/N increases, the number of possible divisions decreases. This is due to the greater importance of search time cost over total cost. For NR/N = .01, .02, .05, we observe that the marginal total cost improvement decreases as we divide the file in several subfiles; this is because at each step the possible storage saving decreases (see Fig. 36).

Thus, within this range, it seems to be always possible to divide the file into 2 and 3 subfiles, but we point out some facts that could lead to a different conclusion at the decision-making level. As the ratio NR/N increases, the first subfile tends to increase in size, as is shown in Table 7. This is because the critical variable is key length of the first group. Storage cost per request decreases inversely to request volume. And minimum total cost per request depends heavily on not having to search the second subfile as request volume grows, which means more of the records are economically pulled into the first group, K_1', as k_i varies. (See eq. 13.)

Table 7: Percent of Records in Each Subfile

NR/N	TWO GROUP DIVISION		THREE GROUP DIVISION		
	I	II	I	II	III
0.01	85.5	14.5	59.0	31.0	10.0
0.02	85.5	14.5	63.6	27.5	8.9
0.05	85.5	14.5	80.6	13.3	6.1
0.10	85.5	14.5	85.5	10.1	4.4
0.20	89.0	11.0	89.0	10.7	0.3
0.50	92.4	7.6	92.4	7.599	0.00041
0.70	93.6	6.4	93.6	6.399	.00041
1.00	94.1	5.9	94.1	5.899	.00041

We can see that for NR/N larger than 0.2, the percentage of records in the third group is so small that for some files it might not be feasible to divide it in three parts. Besides, from Table 6, we find out that the improvement from 2 to 3 divisions for similar NR/N is very small. This result is shown graphically in Fig. 37, which gives the average cost per search as a function of search volume and of the number of files.

4.4 Division of an Index File Based on Usage Frequency

A second consideration in improving the efficiency of an indexed sequential file is the division of the file into two subfiles F'(1), F'(2), one containing the records of high usage frequency, which will be searched first, and the other the records of low usage frequency. Both subfiles are to be stored in the same type of device.

4.4.1 Assumptions

As in the previous analysis, we assume that if the file is large enough to require one or more master index levels, the highest is stored in core. We also assume sequential search beginning in the first subfile, which implies the ability to recognize if a given request has been satisfied in the first file. If this is the case, the search ends; otherwise, we search the second subfile. The following notation is used:

TU = total search time for the undivided file

TD = total search time for a divided file

NR = total number of requests per unit of time

N = total number of records in a file

TB = access motion time

NIL = number of required index levels

TK = data transfer time

K = seek time

Subscript (1) makes reference to the first subfile, (2) to the second subfile.

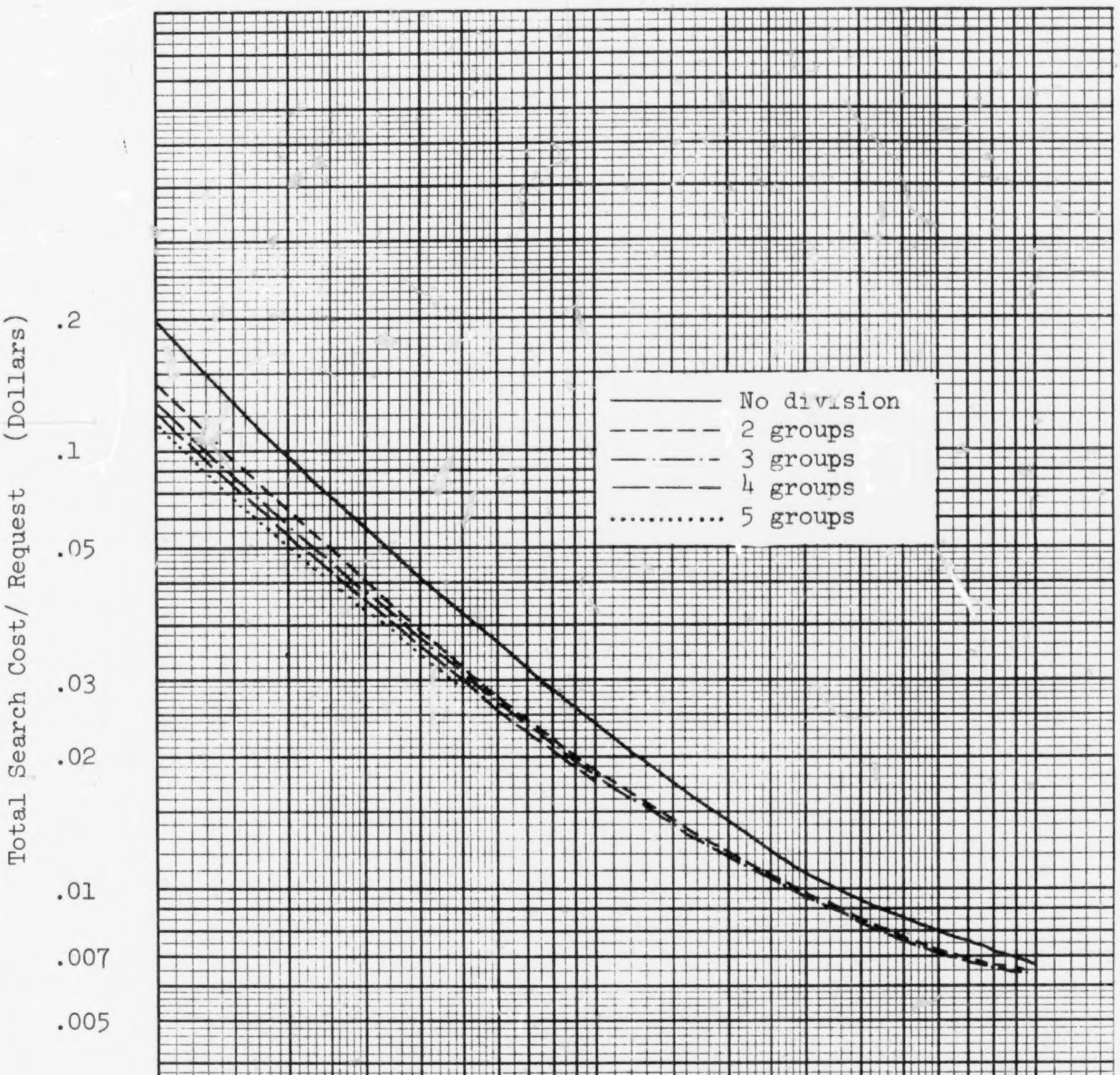
4.4.2 Condition for Feasibility

The total storage requirement is essentially the same, hence only search time will be taken into account.* A division of file F into subfiles F'(1), F'(2) is convenient if TU>TD, or

---

*Assume all file segments are to have equal key length and record lengths, at this stage of analysis.

FIG. 36: COST PER REQUEST AS A FUNCTION OF THE RATIO SYSTEM VOLUME - FILE SIZE FOR FIVE DIFFERENT DIVISION STRATEGIES



$$NR[TB+TK+K]>NR[TB(1)+TK+K(1)] = (NR-NR1)\ [TB(2)+TK+K(2)] \qquad (16)$$

After some algebra,

$$\frac{NR1}{NR} > \frac{K(1) + TB(1) + K(2) + TK + TB(2) - TB - K}{TB(2) + TK + K(2)}, \text{ or}$$

$$\frac{NR1}{NR} > \frac{K(1) + TB(1) - TB - K}{TB(2) + TK + K(2)} + 1 = \alpha \qquad (17)$$

where $0 \le \frac{NR1}{NR} \le 1$

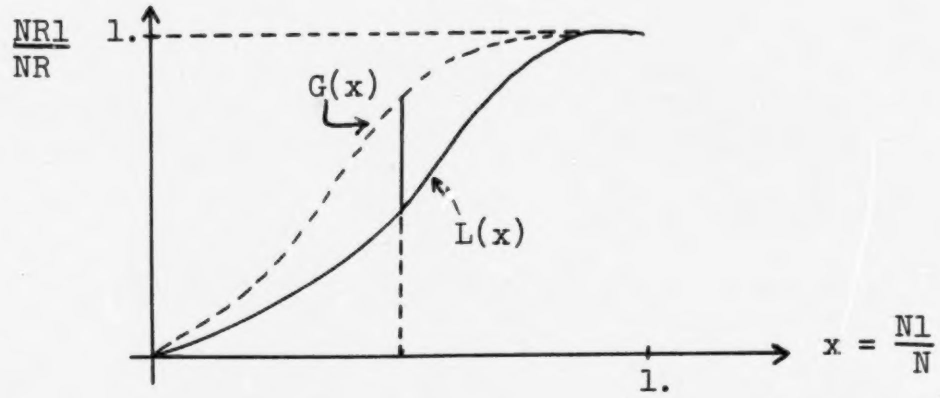
Therefore, if $\alpha>1$, we have a contradiction and no division is possible. We say that a division is feasible if $\frac{NR1}{NR}>\alpha$, and $\alpha<1$.

Because the value of α is only a function of the subfile size N1 and the device's characteristics, we can obtain a one-to-one correspondence between N1/N and α. Then, a necessary condition for dividing a file F into subfiles F'(1), F'(2) is that subfile F'(1) must satisfy a percentage of requests (NR1/NR) larger than α.

4.4.3 Required Distribution

Using the ideas presented above, it is possible to obtain a "Required Cumulative Distribution" L(x), X = N1/N = o,..., 1 as presented in Fig. 37. Assuming we have an actual usage frequency distribution with cumulative function G(x), the optimal division point for a given file F corresponds to an X, for which G(x) - L(x) = Maximum.

FIG. 37:
REQUIRED CUMULATIVE DISTRIBUTION



Observing equation (17), we notice that $a<1$ if $(TB+K)>[TB(1) + K(1)]$. To restate this in prose: $\alpha<1$ if the access motion time plus data transfer time corresponding to the undivided file is larger than the sum of those corresponding to the first subfile. The seek time is given by:

$$K = TB(NIL-1) = ROT(MT2 + MT1 + CT + TI + 1)/2 \tag{18}$$

where

ROT = Rotational delay time for a given device (latency)

$$MT2 = \begin{cases} 1, & \text{if } RM3 \neq 0 \\ 0, & \text{if } RM3 = 0 \end{cases}$$

$$MT1 = \begin{cases} 1, & \text{if } RM2 \neq 0,\ RM3 \neq 0 \text{ or } RM3 = 0 \\ 0, & \text{if } RM2 = 0 \end{cases}$$

$$CT = \begin{cases} 1, & \text{if } RM(i) \neq 0,\ i = 1,\ldots,3 \\ CIT, & \text{if } RM(i) = 0 \text{ for all } i \end{cases}$$

TI = Constant = Track index tracks

CIT = Cylinder index tracks

RM(i) = Master index tracks, i = 1,2,3

From equation (18) we see that seek time K depends on the number and size of the different index levels. Before continuing our analysis, we point out some observations of the growing rate of index levels.

4.4.4 Growing Rate of Indexes

Cylinder Index size (CIT) increases linearly with respect to file size. Using subroutine STOCAP, keyed records with total length of 80 bytes and different N, we obtained the following equation:

$$CIT = 0.0303 + (0.1891).10^{-4}.N \tag{19}$$

If we create a master index when the cylinder index exceeds 4 tracks, the growing rate for the first master index is given by:

$$RM1 + .00384 + (0.6)\ .10^{-6}\ .N \text{ for } N \geq 250{,}000 \text{ records} \tag{20}$$

From equations (19) and (20) we observe that, while we need a file of 250,000 records to create the first entries of master index I, 2 million records are required for filling up the first track and beginning the second track of the master index. Let's consider the following cases.

4.4.4.1 Small Files

F does not need a master index (N<250,000 for RL = 80)

Then: $TB = TB(1)$

$NIL = 2$

$0<CT<4,\ 0<CT(1)<CT$ (21)a,b

$MT2 = 0,\ MT1 = 0$ for F and $F'(1)$

Then recalling the condition, [$\alpha<1$ if $TB + K>TB(1) + K(1)$], it becomes $CT>CT(1)$ from equation (18). (22)

By relations (21)a,b relation (22) is true. Therefore, for small files, such that $NIL = 2$, $a<1$.

4.4.4.2 Large Files

F requires a master index ($2.5\times10^5<N<7\times10^6$).

First subfile size $N1<250{,}000$.

4.4.4.2.1 Case A:

$TB = 0$, $TB(1) \neq 0$, implies the condition $K>TB(1) + K(1)$, for $\alpha<1$ from equation (3), eliminating all constants, and noticing that

$MT2 = 0$; $MT1 = 0$ for F and F'(1)

$CT = 1$; $CT(1) = CIT(1)$

$NIL = 3$ for F, $NIL = 2$ for F'(1), we have that,

$2\cdot TB + ROT(1 + 1)/2>TB(1) + TB + ROT(CT(1) + 1)/2$

$ROT>ROT(CT(1) + 1)/2$

where $0<CT(1)<4$

then $\alpha<1$ for $0<\frac{(CT(1) + 1)}{2}<1$, or $\alpha<1$ for $CT(1)<1$

To restate in prose, $\alpha<1$ only if the first subfile F'(1) is so small that it requires less than one full track of cylinder index.

4.4.4.2.2 Case B:

First subfile size $N1>250{,}000$.

$TB = TB(1) = 0$

$NIL = 3$ for F and F'(1)

$MT2 = 0$; $MT1 = 0$; $CT = 1$ for F and F'(1)

then for $\alpha<1$ we need:

$TB + K>TB(1) + K(1)$, or

$K>K(1)$, but

$2TB + ROT(1 + TI + 1)/2 + 2TB(1) + ROT(1 + TI + 1)/2$

and $K = K(1)$

Therefore, $\alpha=1$, and no division is possible. The results presented above were found consistent with numerical calculations. Using subroutine TSPO, we obtained "Required Cumulative Distributions" for both cases, A and B. Results for case A are presented in Fig. 38 and for case B in Table 8.

FIG. 38:
CUMULATIVE REQUIRED DISTRIBUTION - CASE A

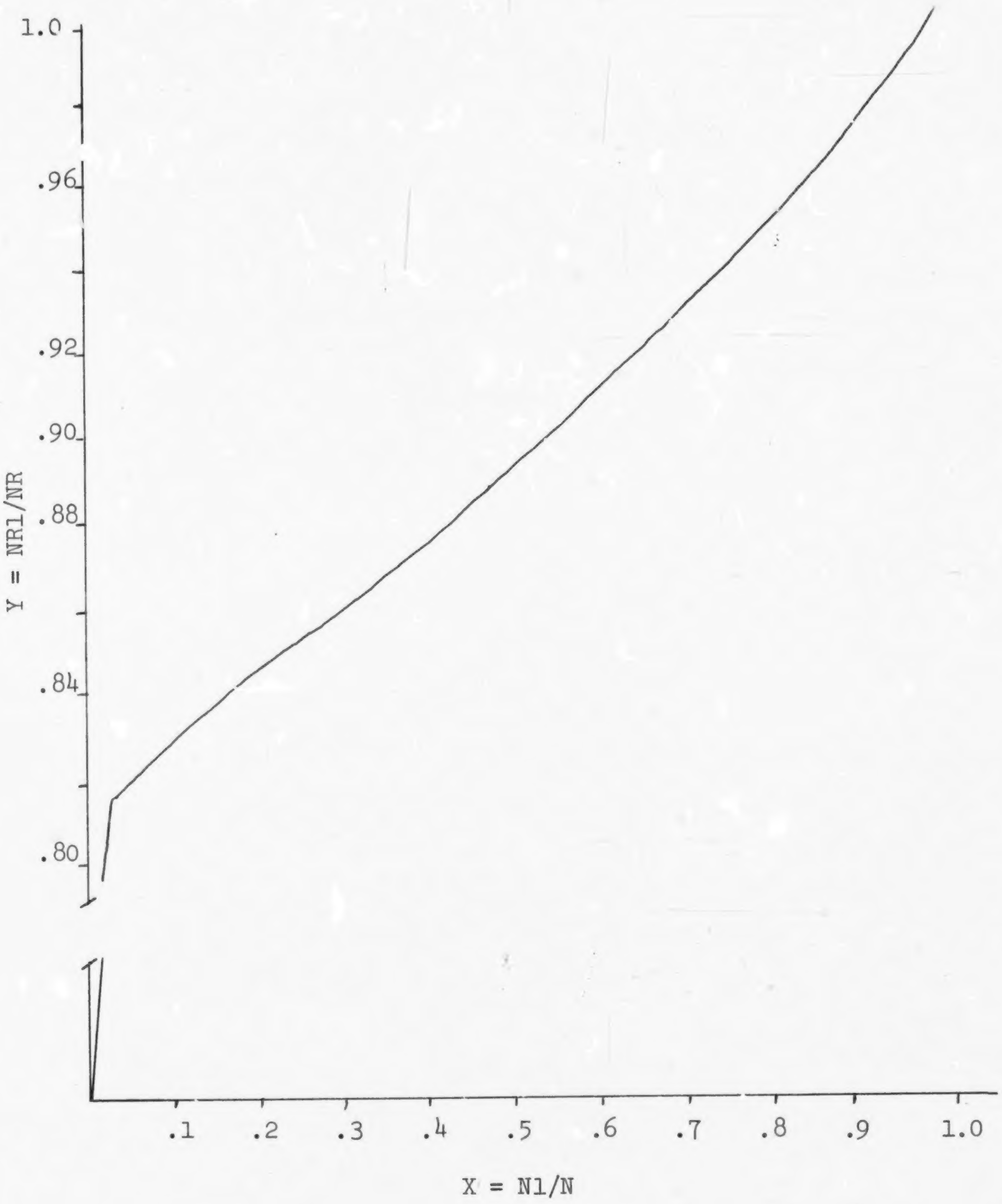


Table 8: Cumulative Required Distribution, Case B

N1/N	C.R.D.
.02	.9545
.04	.9655
.06	.9765
.08	.9875
.10	.9986
.12	1.0096
.14	1.0206
.	.
.	.
.	.
.40	1.1638
.42	1.00
.44	1.00
.	.
.	.
.	.
.98	1.00
1.00	1.00
Record Length: 80 bytes. N = 500,000 R.	

4.4.5 Conclusions

The analysis presented for cases A and B above covers a wide range of file sizes, and the results can be shown to be exact and consistent under the assumptions used. We have no data about real usage frequency distributions presently available, and no reasonable estimation can be made until an experimental system is installed and testing can begin. However, the concept of "required distribution" does enable us to rule out "not possible" cases. Case A (for small files) presents an interesting distribution. (See Fig. 39). We cannot talk of "feasible division points," but it is not unrealistic to say that 70% of the records in a file are used or are required more than 93% of the time. In case B (for large files) we obtain a contradiction for $N1/N>0.1$ or $N1>50,000$, and no division is possible. For $N1<50,000$ or $N1/N<0.1$, $\alpha<1$, but it is unrealistic to assume that 8% (or less) of the records in a file are required more than 95% of the time.

4.4.6 An Extension

A combination of the two kinds of segmentation may be viewed as follows. Suppose a file F which is expected to be used heavily is given and that the set of most frequently requested records is known. Assuming that any subset of the records in the file has a similar key length distribution, F may be divided as follows:

Segment 1. A first subfile containing the records most frequently used. This subfile will probably have a high "Volume of request - file size" ratio, $(\alpha>1)$, and no division recording to key lengths will be possible, in this segment. (See Section 4.3.)

Segment 2. The rest of the records $(F-F_1')$, which will be larger than F_1' and will have a low volume of requests (the ideal condition for a division based on key length).

Therefore, even if the usage frequency distribution for the file F does not meet the "required cumulative distribution," the savings in storage cost of the set of less frequently used records may make the division feasible. (See Section 4.8 for further analysis and interim results of simulation.)

4.5 Estimation of Overflow Areas for Index Files

At this point we turn our attention to one of the maintenance aspects of the index file - updating. In general, the efficiency of ISAM file operation decreases as it is subject to updatings. This is due to the use of overflow areas, "cylinder" or "independent", or both, where the new records are stored. Links have to be established to maintain the logical relationship between old and new records, with corresponding increase in retrieval time. We shall attempt to analyze the problem and make suggestions which hopefully will lead to better performance.

4.5.1 Overflow Records

Under indexed sequential organization, records are stored in lexicographic order on vertically adjacent tracks which form a cylinder. When a new record arrives, it is written in its proper sequential location on the corresponding prime track. The rest of the records are moved up one location, the bumped record is written in the first available location in the overflow area, and the track index entries are properly changed. If the overflow area is located in the same cylinder, no additional seek time is required to retrieve overflow records. The question remains how to distribute the overflow tracks among the cylinders such that the probability of exceeding the overflow area is equal for all the cylinders.

4.5.2 Probabilistic Model

William S. Jewell* has solved a similar problem using order statistics concepts.

---

*Jewell, W.S., "A Filing Problem," Management Science, (8, 1962), [illegible]0-14.

4.5.2.1 New Key Expected Insertion

Let $\{k_i\}$ be the sequence of keys attached to the access records in the file and $\{j\}$ be the set of prime area cylinders. If k_t is the lowest key in cylinder j and k_{t+m} is the highest, the probability of a new record being inserted in cylinder j is,

$$P(j) = P(k_t < k_i \leq k_{t+m}) \tag{23}$$

If N = Total number of new records to be added,

$F(k_t)$ = Probability of a new key less than k_t,

$F(k_{t+m})$= Probability of a key less than k_{t+m},

n = Number of existent records in the file,

$$\text{then } P(C_j) = F(k_{t+m}) - F(k_t) \tag{24}$$

Gumbel* shows that the joint density distribution of the $(k_t)^{th}$, $(k_{t+m})^{th}$ ordered values is

$$Q_n(k_t, k_{t+m}) = CF_t^{\,t-1}(F_{t+m} - F_t)^{m-1}(1-F_{t+m})^{n-t-m} ft \; ft{+}m \tag{25}$$

$$\text{where } C = \frac{n!}{(t-1)!(m-1)!(n-t-m)!} \tag{26}$$

Then the probability that, on the average, x new keys are inserted between the t^{th} and $(t+m)^{th}$ keys, is,

$$w(n,m,N,x) = \int\!\!\int_{0 \leq k_t < b_{t+m} \leq \infty} \mathscr{P}_n (k_t, k_{t+m}) P(N,m,x \mid k_t, k_{t+m}) dk_t \; dk_{t+m}$$

which Jewell shows is equal to the distribution of exceedances

$$w(n,m,N,x) = \left(\frac{m}{N+n}\right) \frac{\binom{n}{m}\binom{N}{x}}{\binom{N+n-1}{x+m-1}} \tag{27}$$

$$\text{with a mean value of, } \overline{x}(n,m,N,x) = \left(\frac{m}{n+1}\right) NN \tag{28}$$

4.5.2.2 Estimation of Overflow Areas

From (27) and (28) it is clear that w depends only on the original number of records in the file n, on the interval m between the lowest and highest key in the cylinder, and on the number of new records N. The probability that the overflow area of cylinder j, (S_j), is exceeded is given by:

$$W(S_j) = \sum_{x=S+1}^{N} w(n,m,N,x) \tag{29}$$

The probability that the overflow area of cylinder j is not exceeded is

---

*Gumbel, E.C., Statistics of Extremes, New York: Columbia University Press, 1958, p. 57-74.

$$V(S_j) = \sum_{x=0}^{S} w(n,m,N,x) = 1-W(S_j) \qquad (30)$$

Let A_j be the event that the overflow area of cylinder j is exceeded and assume the A_j's are independent; then $W(S_j) = P(A_j)$ and

$$\hat{W}(F) = P(\bigcup_{j=1}^{CL} A_j) = \sum_{j=1}^{CL} W(S_j) - \sum_{j=1}^{CL} W(S_j) \qquad (31)$$

is the probability that an independent overflow area is used. CL is the total number of cylinders. Recalling that

$$V(S_j) = P(A_j^c) \qquad \text{with} \qquad (\bigcup_{j=1}^{CL} A_j)^c = \bigcap_{j=1}^{CL} A_j^c, \qquad (32)$$

$$\text{then } V(F) = P(\bigcap_{j=1}^{CL} A_j^c) = \prod_{j=1}^{CL} V(S_j)$$

is the probability that none of the cylinder overflow areas is exceeded. From (27), w (·) is equal for similar intervals m̲. Hence, if m is the same for all cylinders, equations (31) and (32) can be expressed as

$$\hat{W}(F) = CL \cdot W(S_j) - [W(S_j)]^{CL} \qquad (33)$$

$$\hat{V}(F) = [V(S_j)]^{CL} \qquad (34)$$

If the ratio N/(n+N) converges to a value g, w(·) in equation (27) can be approximated by

$$\hat{w}(m,x) \approx \binom{m-1+x}{m-1} g^x (1-g)^{m-1} \qquad (35)$$

If Q is the cylinder capacity, then CL = n/(Q-S), and m = Q-S, where Q and S are expressed as records per cylinder. If we know N, equations (29), (30), (33), and (34) depend on the value of S, and V(**F**) in equation (34) can be used as a tolerance limit. As S increases, $V(S_j)$ and V(F) increase, and we can find a cylinder overflow capacity S that yields a probability $\alpha = \hat{V}(F)$ that no independent overflow area is needed. Thus, it has been proved that the best policy to follow in the allocation of cylinder overflow areas is that of assigning the same number of overflow tracks to each cylinder.

4.5.3 Alternative Estimation

Another kind of tolerance limit can be achieved by using a value of N obtained by testing a null hypothesis on its true value. This is justified by the fact that N will certainly behave as a random variable. Then if N* is a number such that,

$P(N \leq N^*) = \alpha$, the cylinder overflow area S is given by:

$S = Q \cdot N/(N + \frac{ORT}{NRT} \cdot n)$ (overflow records per cylinder) or, expressed as overflow tracks per cylinder IOTC,

$$IOTC = N \cdot TPC/(N + \frac{ORT}{NRT} . n)$$

where, Q = cylinder capacity
ORT = overflow records per track
NRT = normal records per track
TPC = number of tracks per cylinder

4.6 Allocation of Non-Keyed Records to Storage Blocks

So far we have presented two methods of improving the performance of the index file and suggested a way of combining both. In what follows, we introduce a general procedure to allocate non-keyed records to blocks, that may be applied either to the pointers file, or to the master file of variable-length records. The next section deals with the estimation of overflow areas for the pointers file, which we show is a different case from that of the index file.

A non-keyed file using the direct (random) organization is not system-controlled and allows the user almost complete freedom of design. In general, non-keyed master records are variable length and blocked to full track size (FTB). If the records are allocated to the blocks at random, it is likely that often the last record in the block is split between two consecutive blocks.* When retrieving such a record, it is necessary to read both blocks (tracks) and when the file is heavily used, its overall performance will be affected. Therefore, it is desirable that all records fit exactly in the blocks such that no record is split across track boundaries.

4.6.1 Mathematical Model

Let R be the set of all records r, and Ri be the subset of records of R with identical length i. Obviously, all Ri are disjoint and R = $\sum_i$ Ri.

Let B be the set of blocks bj, j = i,...,NB. We assume that all blocks have equal length, for otherwise there would be no problem. NB is calculated using the average record length.

If we consider each subset Ri as a node i called a source, having a capacity equal to the number of records which belong to the subset, say Ci, and each block bj as a node j called sink with capacity t, and if we let S = {i} be the set of sources and T = {j} be the set of sinks, we can formulate the problem as a flow problem with gains. (See Fig. 40.) The concept of "gains" is related to the fact that if we allocate X records r_i to a block j, the capacity of source i is now (Ci - X), while the capacity of block j is (t - i·x). The X records from source i are multiplied by (i). Hence, we define multipliers π_i^j = i. We state the problem as follows:

Let G = (S,T,A,) be a bipartite graph, where S = {i} is a set of sources, T = {j} the set of sinks and A = {(i,j)} the set of arcs connecting sources and sinks.

---

*Splitting of Pointer File records can easily be precluded. They are short, fixed length and should be blocked to a blocksize which itself is evenly divisible into disposable track capacity.

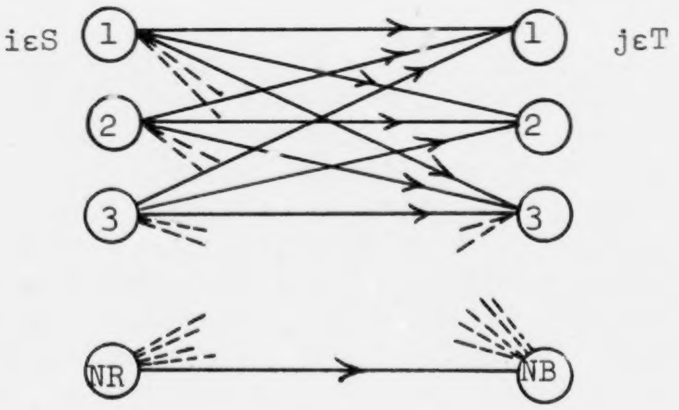
$$\text{Maximize} \sum_{ij} Xij \qquad (36)$$

$$\text{Subject to } \sum_j Xij \le Ci, \; i = 1,\ldots.,NR \qquad (37)$$

$$\sum_i \pi_i Xij \le tj, \; j = 1,\ldots.,NB \qquad (38)$$

$$Xij \ge 0; \; i = (i), \text{ for all } i,j.$$

FIG. 39:
NETWORK REPRESENTATION



Each node (i) in the figure corresponds to a different record size (i), its capacity Ci corresponds to the number of records with size i; each node j corresponds to a different block j. We say that the flow in arc (i,j) is larger than zero (Xij>0) is one or more records of size i are allocated to block j. Then it is obvious that if we find a set of Xij's which satisfy (37) and (38) with the equality sign, we have allocated all existent records to the different blocks in such a way that no space is wasted. Moreover, we can show that is (37) and (38) are expressed in equality form, and

$\sum_i \pi i \; Ci = \sum_j tj$ holds, the problem always has a solution. (39)

In fact, assume we have found a set of Xij that satisfies

$\sum_i Xij\pi i = tj$ for all tj's, then

$$\sum_j tj = \sum_j \sum_i Xij\pi i = \sum_i \pi i \sum_j Xij = \sum_i \pi i \; Ci \qquad \text{by } (40)$$

Therefore, $\sum_j Xij = Ci$ for all i's Q.E.D.

A proof beginning with $\sum_j Xij = Ci$ follows a similar argument.

4.6.2 Inefficiency of Random Allocation

Let us estimate the probability of failing to obtain an optimal packing if the records are allocated to the blocks at random. The problem given by (37) and (38) is a linear programming problem of the type

$$AX \leq b \qquad (41)$$

where A is an (ND + NT) x (ND.NT) matrix. Because of the multipliers πi, the rank of A is (ND + NT). The set of solutions X to (41) are contained in a ND.NT) - dimensional vector space. A feasible solution X to (41) forms a convex polyhedral cone C spanned by (ND + NT) independent vectors. Therefore, the number of feasible solutions X to (41) is equal to the number of ways we can choose (ND + NT) vectors out of (ND.NT), namely,

$$S = \binom{ND.NT}{ND+NT} \qquad (42)$$

Let us now calculate the total number of possible allocations of records to tracks. Assume that all blocks are placed one after the other forming a big block MB divided in NT parts, by NT - 1 division lines. By equation (40) the length of MB is

$$LMB = \sum_j bj = \sum_i \pi_i \; Ci \qquad (43)$$

and $\sum_i Ci = K$, the total number of records.

The packing is optimal if no record intercepts any division line. We can see that the total number of ways K records can be allocated to MB is K! Therefore, the probability of obtaining an optimal random packing is:

$$P^* = \frac{\binom{ND.NT}{ND+NT}}{K!} \qquad (44)$$

For a small example with K = 75, ND = 11, NT = 5, we have

$$P^* = \frac{55!}{39!\ 16!\ 75!} = \frac{(1.26)10^{75}}{(2.09)10^{13}\ (2.03)10^{46}\ (2.4)10^{109}}$$

It is clear that P* is indeed very small.

4.6.3 Computational Tool

Subroutine BLOCKING (MD, LSD, ND, KM, NT) has been developed to solve the allocation problem. The inputs are:

MD = A (ND × 2) matrix, such that MD (i,1) is the record length, and MD (i,2) the number of records with the same length

NUL = Block capacity

ND = Number of different record lengths

KM = A large number≈500

NT = etc.

The elements of matrix MD are ordered (beginning with the smallest), and all the records are assigned a number (beginning with the largest). The output consists of an enumeration of records allocated to each block.

4.7 Estimation of Overflow Areas for the Pointers File

Let Pi be the subset of pointers pj mapping into an access record ai. A subset Pi may be considered as a logical record because all $pj \in \bar{P}i$ are contiguous in the pointers file in a physical sense. To simplify notation, let $ri = \bar{P}i$. The set of address records $\{r_i\}$ has a distribution G(i), which we assume remains constant over time. When new master records are added to the master file, two things can happen to the pointers file: a) a new record r_i is added, or b) a new pointer p_j is added to an existing record, r_i. The first case does not present difficulties and the new r_i is located at the end of the pointers file. It implies the addition of a new access record.

4.7.1 Former Scheme

In the second case the following procedure has been suggested:

Given ri = {pj;j = 1,...,k} and pk + 1

a. Move pk from its original location and write it together with pk + 1 on an overflow area

b. Write the new address of pk on its original location

c. Change the count field of the access record ai to indicate the addition of pk + 1.

Two difficulties are evident. First, if *n* new pointers are added to a record, the total storage used is (n + 1) times the pointer's length. Second, the retrieval time for a record with additions increases by at least a rotational time (when the new pointers are located on a cylinder overflow area) or by an access motion time plus rotational time (when the new pointers are written in an independent overflow area).

4.7.2 New Scheme

If, instead of using cylinder overflow areas, we define *track* overflow areas when a new pointer is added to any of the records in the track, it can be written contiguous to the proper record with the other records being moved one place toward the end of the track. In this form, the retrieval time of the record is not increased, and no additional link is used. There still remains the problem of estimating the size of track overflow areas. Also, if at any of the tracks of a given cylinder, the overflow area is exceeded, a cylinder overflow area becomes necessary.

It should be noticed that the first scheme mentioned above is similar to that used for the index file. We allocate some overflow tracks to each cylinder, and if one or more are exceeded, an independent overflow area is used. In the second case, the cylinder overflow area plays the same role as the independent overflow area. The better efficiency of the second method is obvious.

4.7.3 Differences

Let us point out the differences between the pointers file and index file cases. We say that an address record is in "state i" if it consists of (i) pointers. If such a record receives an addition we say that it changes to state (i + 1). We shall use consecutive letters to denote consecutive states.

a. Index records ai have equal length L. Hence, the total storage required is $L\sum ai$. Address records in state (i) have length (i) and the total storage required is $\sum in_i{}^t$, where $n_i{}^t$ represents the number of records in state i.

b. An addition to the index file is a record a_j which is inserted in its proper sequential location with respect to the existing a_i's. Additions to the pointers file are pointers p_j which change the state of the corresponding records in such a way that the resulting set of $[r_i]$ have the same distribution. If the records r_i were in order, after a set of additions, the order is destroyed.

c. For the index file, the overflow area needed is proportional to the size of the subset of records already stored in a cylinder. For the pointers file the overflow area is proportional to the number of records which are likely to change of state.

4.7.4 Stochastic Model

4.7.4.1 Assumptions

We assume that the pointers file constitutes a system governed by a Markov process defined by a finite set of states and a matrix of transition probabilities. Furthermore, we assume that the process is stationary and the state of largest value is persistent over a reasonable period of time.

The matrix of transition probabilities is a stochastic matrix Q = $||qi_j||$, such that,

$$qi_j \begin{cases} \geq C & \text{for } j \geq i \\ = 0 & \text{for } i < j \end{cases}$$

$$\sum qi_j = 1$$

$$q_{nn} = 1; \; n = \text{largest state.}$$

Let n^t be the total number of records in the file at time t. n^t is distributed as G(i), and $n_i^t = g(i)n^t$ is the number of records in state i at time t. The total number of pointers in the file is $p^t = \sum j n_j^t$. The total number of new pointers to be added to the file at the end of period t is N^t. We assume that,

$$\frac{N^t}{P^t+N^t} \approx C, \text{ as } P^t, N^t \to \infty. \tag{45}$$

4.7.4.2 Proof of Stationarity

We shall show that for a simplified case in which the probability of two or more simultaneous additions is zero, the process is indeed stationary under the assumption of constant distribution G(i).

$$P^{t+1} = P^t + N^t \tag{46}$$

$$\text{by 1, } N^t = \frac{C}{1-C} P^t \tag{47}$$

The average record length L, is constant for all periods t.

$$L = \sum_j j\cdot g(j) = \frac{\sum j\ n_j^t}{\sum n_j^t} = \frac{P^t}{n^t} \tag{48}$$

combining (46) and (48) we obtain,

$$n^{t+1} = n^t \left(1+\frac{N^t}{P^t}\right) = n^t \left(\frac{1}{1-C}\right), \text{ and} \tag{49}$$

$$n_j^{t+1} = n_j^t \left(1+\frac{N^t}{P^t}\right) = n_j^t\left(\frac{1}{1-C}\right) \tag{50}$$

Recalling that the expected value of the number of exceedances is given by, $\overline{X} = N\cdot M/(n+1)$, we can say that the expected number of records in state j at period t+1 is equal to the number of records in state j at period t plus the expected number of new records in state j.

$$n_j^{t+1} = n_j^t + \frac{n_j^t R^t}{n^t+1}$$

where $R^t = n^t \left(\frac{C}{1-C}\right)$ is the number of new records.

$$\text{Then, } n_j^{t+1} = n_j^t \left[1+\frac{N^t}{P^t}\left(\frac{n^t}{n^t+1}\right)\right] \tag{51}$$

Hence, the results obtained in equation (50) are equal to those given by (51) for n^t large enough, such that

$$\frac{n^t}{n^t+1} \approx 1 \tag{52}$$

This implies, as may be expected, that the behavior of the number of records in state j as a function of time, follows the order statistics concepts presented before. If X^t_{jk} is the number of records in state j that change to state k at the end of period t, then equation (53) is the probability that a record in state j will change to state k at the end of period t.

$$q^t_{jk} = \frac{X^t_{jk}}{n^t_j} \qquad (53)$$

$$\text{Using (50), } q^t_{jk} + 1 = \frac{(1-C)\ X^{t+1}_{jk}}{n^t_j} \qquad (54)$$

It is obvious that $q^{t+1}_{jk} = q^t_{jk}$, if $X^t_{jk} = (1-C)\ X^{t+1}_{jk}$.

We shall prove this recursively, beginning with the largest state n. By the assumption of persistency, $q^t_{mn} = \frac{X^t_{m_n}}{n^t_m}$

$$\text{and} \qquad X^t_{mn} = n^{t+1}_n - n^t_n. \qquad (55)$$

Using equations (50) and (55) we obtain,

$$q^{t+1}_{mn} = \frac{n^t_n}{n^t_n}\ \frac{C}{1-C} = q^t_{mn}$$

$$\text{and } X^{t+1}_{mn} = \frac{X^t_{mn}}{1-C} \qquad (56)$$

$$\text{Proceeding backwards, } n^{t+1}_m = n^t_m + X^t_{1m} - X^t_{mn}, \qquad (57)$$

$$\text{by (50), } X^t_{1m} = n^t_m \frac{(C)}{1-C} + X^t_{mn}, \qquad (58)$$

$$X^{t+1}_{1m} = n^{t+1}_m \frac{(C)}{1-C} + X^t_{mn},$$

And using equations (50) and (56), $X^{t+1}_{1m} = \frac{X^t_{1m}}{1-C}$

Equation (57) is general enough to conclude that equation (58) holds for any consecutive pair of indices (ij).

4.7.4.3 Distribution of Xij

Let us give more attention to equation (58). Replacing C for its value, we obtain:

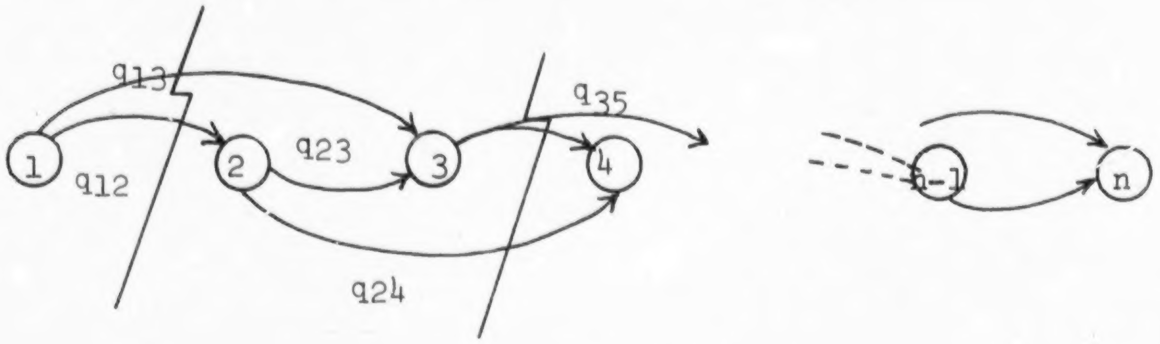
$$X_{ij}^{t+1} = X_{ij}^{t} \left(1 + \frac{N^t}{P^t}\right)$$

which is similar to (50) and suggests that the number of transitions (ij) at time t + 1, is equal to the number of transitions (ij) at time t plus the expected number of new transitions (ij). Under the assumption of one step transition for which (58) was obtained, we find that

$$X_{ij}^{t+1} = X_{ij}^{t} \left(1 + \frac{N^t}{P^t+1}\right).$$

This leads to the conclusion that X_{ij}^{t} has an unknown distribution F(ij). Then, it is possible to predict the value of n_j^{t+1} and if F(ij) is known, also the value of X_{ij}^{t+1}. However, the first element alone is not enough to estimate the required overflow areas, and if, as is expected in a real case, the transition probabilities for more than one step are not equal to zero, it may be impractical to estimate F(ij). In order to appreciate the complexity of the pointers file behavior, we introduce a simple example.

FIG. 40:
REPRESENTATION OF TRANSITION PROBABILITIES



Any stochastic matrix may be expressed as a graph. The nodes represent the states of the system and the arrows the possible transitions. For instance, any element in state 2 has a probability p_{23} of going to state 3, and a probability p_{24} of going to state 4. We notice that this case is more general than the case used to prove the stationarity of the transition probabilities. Let us assume that the graph in Fig. 40 represents the pointers file and at time t, all the records in state 2 and 3 are stored in the same track. After updating at the end of the period t, the amount of storage used in the track is given by,

$$S^1 = 2q_{22}n_2^t + 3\,(q_{33}n_3^t + q_{23}n_2^t) + 4\,(q_{24}n_2^t + q_{34}n_3^t) + 5\,q_{35}n_3^t \qquad (59)$$

We may be tempted to say that similarly to the index file, the amount of overflow area used in a track depends only on the number of records already present in the track. If this was true, by equation (51) and assuming that n^t is large, the amount of storage used in the track would be,

$$S^2 = 2n_2^t \, (1 + \frac{N^t}{P^t}) + 3n_3^t \, (1 + \frac{N^t}{P^t}) = 2n_2^{t+1} + 3n_3^{t+1};$$

$$S^2 = 2(q_{22}n_2^t + q_{12}n_1^t) + 3 \, (q_{33}n_3^t + q_{13}n_1^t + q_{23}n_2^t) \qquad (60)$$

Then S^1 given by (59) is equal to S^2 only if,

$$4 \, (q_{24}n_2^t + q_{34}n_3^t + 5q_{35}n_3^t = 2q_{12}n_1^t + 3q_{13}n_1^t. \qquad (61)$$

Equation (61) introduces a condition on either the distribution G(i), or the stochastic matrix, or both. Moreover, it is possible to visualize that condition (61) depends on the specific physical allocation of the records to the tracks. Therefore, we conclude that in general (61) is not true, and the expected overflow area is not properly estimated by equations similar to (60).

4.7.5 Estimation of Overflow Space

The proposed method for estimating the overflow is the following. Equation (59) can be expressed also as,

$$S^1 = 2n_2^t + 3n_3^t + q_{23}n_2^t + q_{34}n_3^t + 2 \, (q_{24}n_2^t + q_{35}n_3^t)$$

$$S^1 = n_2^t \, (2 + q_{23} + 2q_{24}) + n_3^t \, (3 = q_{34} + 2q_{35}), \qquad (62)$$

and in general, if at time t a given track contains records in states, k,...,n.,. The expected storage used at time t + 1 is given by,

$$S = \sum_{i=k}^{n} n_i \, (i + j \sum_{j=1}^{j} q_{i,\, i+j}) \qquad (63)$$

The expression in parentheses may be regarded as a multiplier that can be estimated using the stochastic matrix Q. An easier method to estimate this multiplier is to use only q_{ii} for all possible states, and an average "transition step" T. Then,

$$S = \sum_{i=k}^{n} n_i \, (i + T \, (1 - q_{ii})) \qquad (64)$$

Once the set of multipliers are evaluated, the records can be allocated to the pointers file using a modified subroutine BLOCKING that accepts real multipliers (as well as integers).

4.8 Multiple Device Systems

4.8.1 General

The operational characteristics of a multiple device system are determined by a set of factors, among which one finds the following:

a. File size, system volume, average record length and average number of master records mapping into an index record.

b. Device capacity, access motion, rotational delay and data transfer times.

c. Control unit used.

It is difficult to state in general which of the factors are more important than others and how they interact. Any priority relationship will probably change for different cases. However, under some assumptions one may be able to analyze the alternatives and draw some conclusions.

4.8.2 DASD Control Units

4.8.2.1 Device Combinations

There are three control units for the different devices. The 2314 storage facility has its self-contained control unit which handles up to eight disk packs or access mechanisms. The 2301 drum uses the 2820 control unit which controls up to four 2301's. The control unit for the other four devices is the 2841 Storage Control which in theory can control up to eight access mechanisms in any combination, but it is actually subject to certain restrictions. These restrictions may be stated as follows:

a. If no 2303 is attached, any combination of 2311's, 2321's and 2302's, up to eight may be attached.

b. If one 2303 is installed, a maximum of three 2311's may be attached.

c. If two 2303's (maximum) are attached, no 2311's may be attached.

In cases b and c any combination of 2321's and 2302's, such that the total number of access mechanisms is eight, is permitted.

4.8.2.2 Device Selection

When considering a multi-device system, it is important to exploit the opportunities of parallel operation. From the brief description of the control units presented above, it follows that some devices present more advantages than others. The 2314 is the best suited for this kind of operation, especially for medium and large files where the data is likely to fill up many or all eight of the disk packs.

The 2301 drum presents the disadvantage that it can be used only with the 2820 control unit and therefore, if less than four 2301's are needed, the operational cost will increase. The 2302 disk and the 2311 disk have some features similar to the 2314, but cannot compete with its operational characteristics. Thus, we shall consider three devices as possible candidates for a multi-device system: the 2314 disk drive, 2303 drum, and 2321 data cell. The latter two would be attached to the same 2841 control unit. The 2321 data cell is a very slow device with a huge storage capacity, but considering that a large percentage of records in a file are seldom used, one may expect that the overall system performance won't be affected.

Using the same approach as in the previous parts of this report (one-device system), we shall consider the division of a file in two or more subfiles, each of them stored in different devices, under the assumption that only one request is processed at a time. Later we shall present some ideas on parallel operations.

4.8.3 Cost Comparisons

In order to give an idea of the operational ranges for the three devices mentioned above, we present a set of cost curves defined over a wide range of the ratio "volume of requests/file size" (NR/N). These curves are presented in Figs. 41 and 42.

As may be expected, for low values of NR/N, the 2321 data cell presents lower costs than the 2314. The break-even point for these two devices was found at NR/N = 0.05. This figure should be interpreted only as a rough estimate of the true break-even point. On one hand, it has been assumed that the CPU-time cost is charged for the whole search time (clock time) and on the other, no parallel operation has been assumed. The second consideration places the 2314 storage facility in a less favorable position. The reason for this statement is the following. The 2321 data cell has a storage capacity which is about 13.7 times larger than a 2314 pack; for each 2314 pack, there is an access mechanism and thus for the same file size, if possible, one could use more than one 2314 access mechanism (a.m.) while still having only one 2321 a.m. Therefore, if parallel operation were to be considered, the break-even point for the 2321, 2314 would shift to the left.

The 2303 drum presents very high cost all along the range of NR/N covered by the graphs. The break-even point for the 2303 and the 2321 is shown at NR/N = 2.1. Even for the ratio NR/N = 10, the 2303 drum presents a higher search cost per request than the 2314 storage facility. This means that an economic operation of the 2303 drum is only possible for a range of very high NR/N ratios.

4.8.4 Vertical Division of a File

Vertical segmentation of a file is the division of it into two or more subfiles, the first of which contains the set of records most frequently requested. The idea is similar to that of "Division of an Index File Based on Usage Frequency" presented in Sec. 4.4 above,

FIG. 41:
TOTAL SEARCH COST PER REQUEST AS A FUNCTION OF THE RATIO
SYSTEM VOLUME-FILE SIZE, FOR DIFFERENT DEVICES

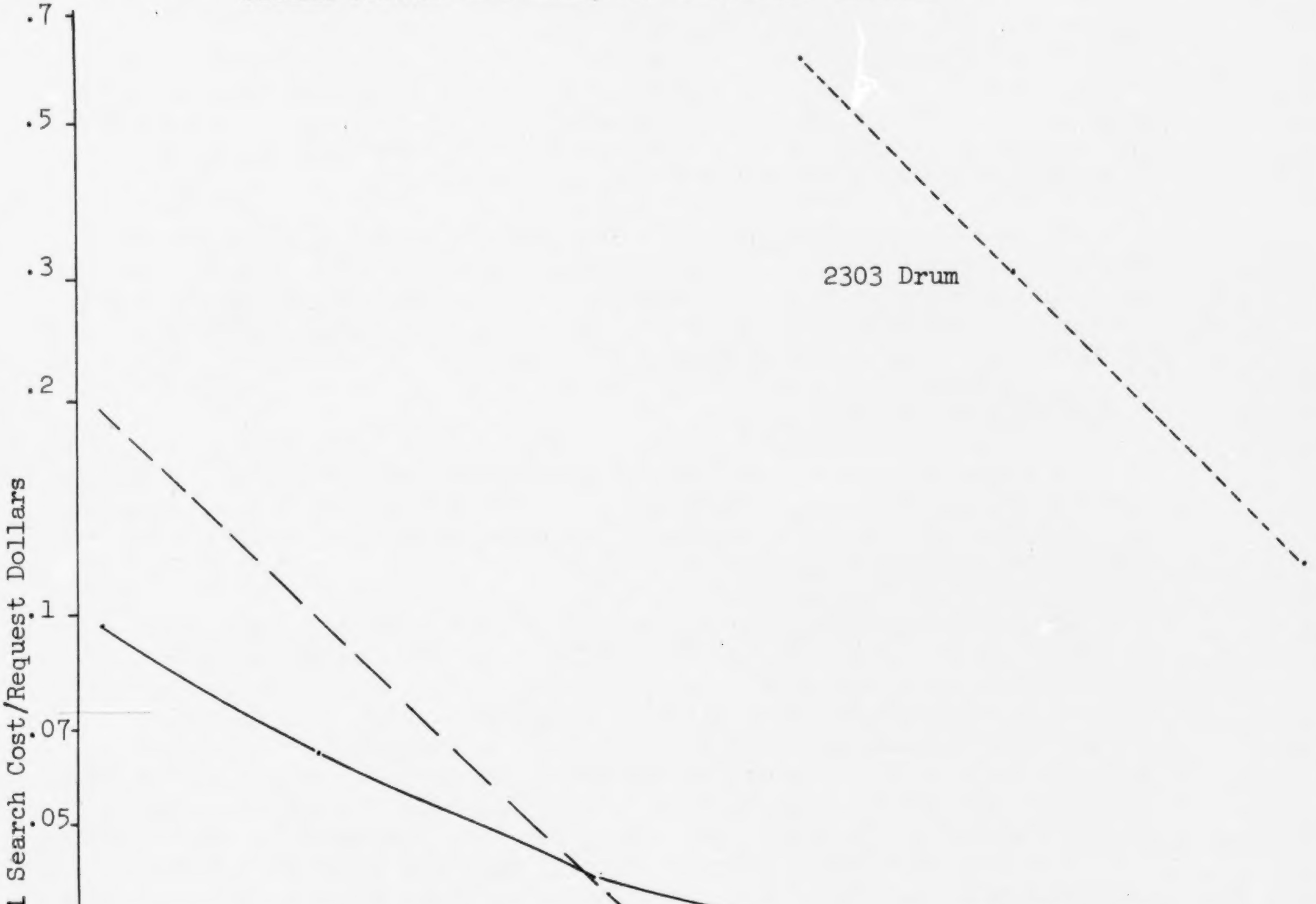
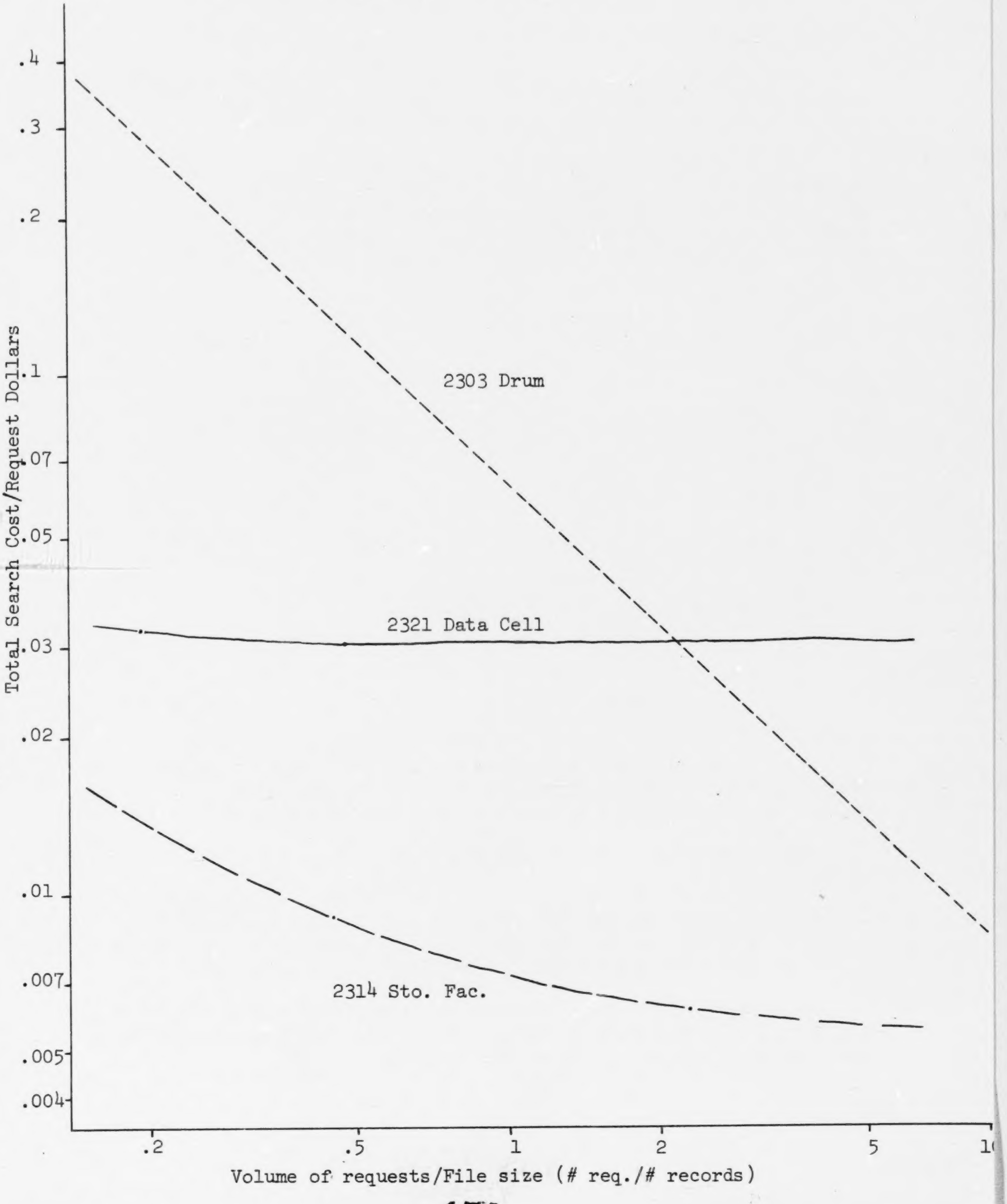


FIG. 42:
TOTAL SEARCH COST PER REQUEST AS A FUNCTION OF THE RATIO SYSTEM VOLUME-FILE SIZE, FOR DIFFERENT DEVICES



but unlike that case, two devices are considered here, and the different storage costs and other factors have to be taken into account.

4.8.4.1 Usage Frequency Distributions

When considering vertical segmentation of a file, we are assuming implicitly the existence and knowledge of a usage frequency distribution Such a distribution can be expressed in different forms.

Let us assume that during a time period of a month, one has observed the distribution of requests over the set of records. A histogram or frequency distribution may be constructed by taking as class marks the number of requests and as frequencies the number of records which have received the same number of requests during the month.

FIG. 43:
USAGE FREQUENCY DISTRIBUTIONS

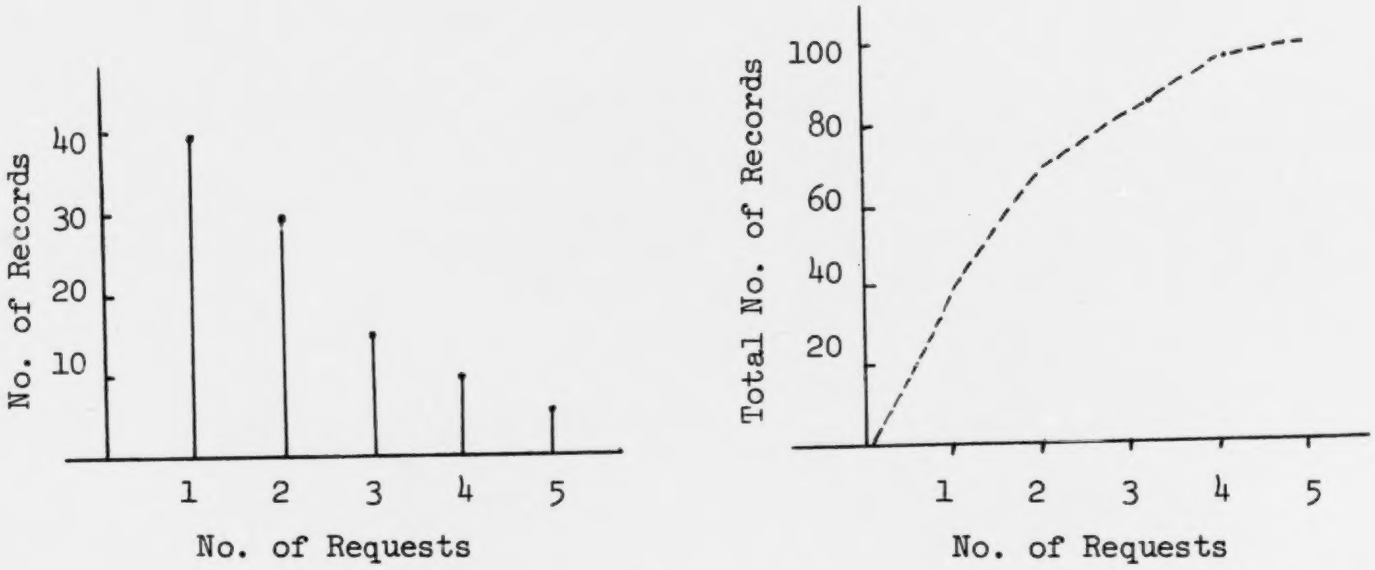


Fig. 43 shows such a histogram and one can observe that both frequency and cumulative distributions have the same range, and the moments of the distribution can be calculated from them.

The original data can be classified as follows:

Xi :	1	2	3	4	5	(No. of requests)
fi :	40	30	15	10	5	(No. of records)
Xifi:	40	60	45	40	25	(Total no. of requests)

$\Sigma fi = T = 100 \quad \Sigma Xifi = n = 210$

If we re-order fi from smallest to largest and define $Z(i) = fi/n + Z(i-1)$ and $Y(i) = Xifi/n+Y(i-1)$ where Xifi has also been re-ordered as the corresponding fi, we obtain the following table.

Zi :	0.05	0.15	0.30	0.60	1.0
Yi :	.119	.309	.525	.810	1.0

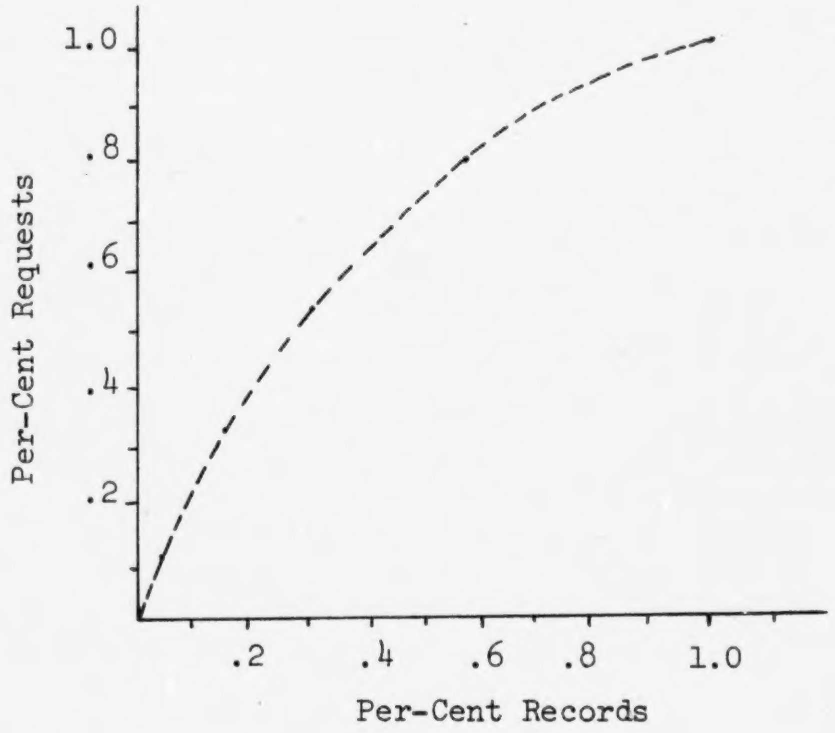
Zi = cumulative percent of records

Yi = cumulative percent of requests

The graph of Zi vs. Yi constitutes a "normalized usage frequency distribution" and is presented in Fig. 44.

Hereafter it is assumed that any frequency distribution is expressed as a normalized cumulative distribution.

FIG. 44:
NORMALIZED USAGE FREQUENCY DISTRIBUTION



4.8.4.2 Cost Equations

The total search cost per request for a two-device file is given by equation (65), and for an undivided file by equation (66), where

Si = storage used by the ith subfile

Ci = cost of storage for the ith subfile

Ti = search time per request for the ith subfile

CT = cost of time

ST = storage used by the undivided file

Tu = search time/request for the undivided file

$F(X_1)$ = percentage of total requests satisfied in a fraction X_1 of records stored in the first subfile.

$$CD = (S_1C_1+C_2S_2)/NR+CT(T_1+(1-F(X_1))T_2) \tag{65}$$

$$CU = STC_1/NR+TuCT \tag{66}$$

4.8.4.3 Equivalent Problems

As in the case of one-device systems, one can say that it is convenient to divide the file if the search cost per request for the divided file is less than the cost for the undivided file using the faster device, i.e.

$$CU>CD. \tag{67}$$

Noticing that CU is a constant for a given set of data, this is equivalent to finding the optimum first subfile size X1 for which CD is a minimum.

Using the first approach, the problem is to find the optimum X1, such that the positive difference (CU - CD) is a maximum.

Replacing equations (65) and (66) in (67), we want to obtain,

$$\text{Max} \left\{F(X_1) - \left[\frac{1}{T_2}(T_1+T_2-TU) - \frac{1}{T_2CT}\left[\frac{C_1(ST - S_1) - C_2S_2}{NR}\right]\right]\right\} \tag{68}$$

4.8.4.4 Required Cumulative Distributions

The function $F(X_1)$ in equation (68) is given by a cumulative usage frequency distribution, live data for which at the present time are unknown. However, calling the expression within the first square brackets $G(X_1, NR)$, it is possible to generate "required cumulative distributions" (RCD) for different values of system request volume NR. The meaning of these RCD's is the same as for the one-device case, i.e., a file is divided only if $F(X_1) - G(X_1, NR)>0$, and the optimal division point is given by

$$\text{Max} \left\{F(X_1) - G(X_1, NR)\right\}.$$

Figures 46, 47, and 48 present sets of RCD's for different large index file sizes: 250K, 500K, and 750K records, respectively. The curves at Figures 47 and 48 have points at which their slopes are negative. Those points correspond to first-file sizes at which the first master index level is created.

FIG. 45:
CUMULATIVE - REQUIRED DISTRIBUTION FOR A
TWO-DEVICE FILE (2314-2321)

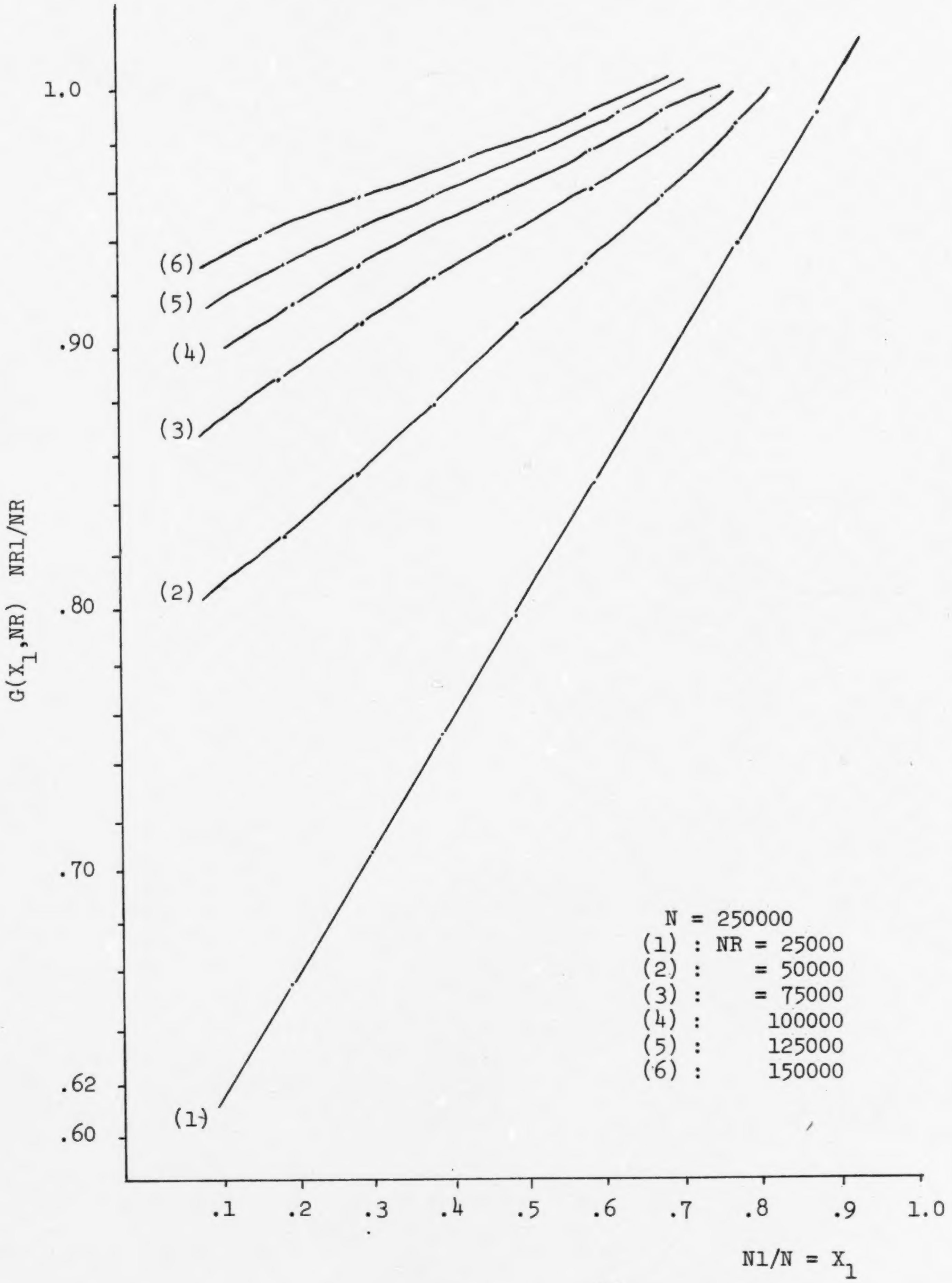


FIG. 46:
CUMULATIVE - REQUIRED DISTRIBUTION FOR A
TWO-DEVICE FILE (2314-2321

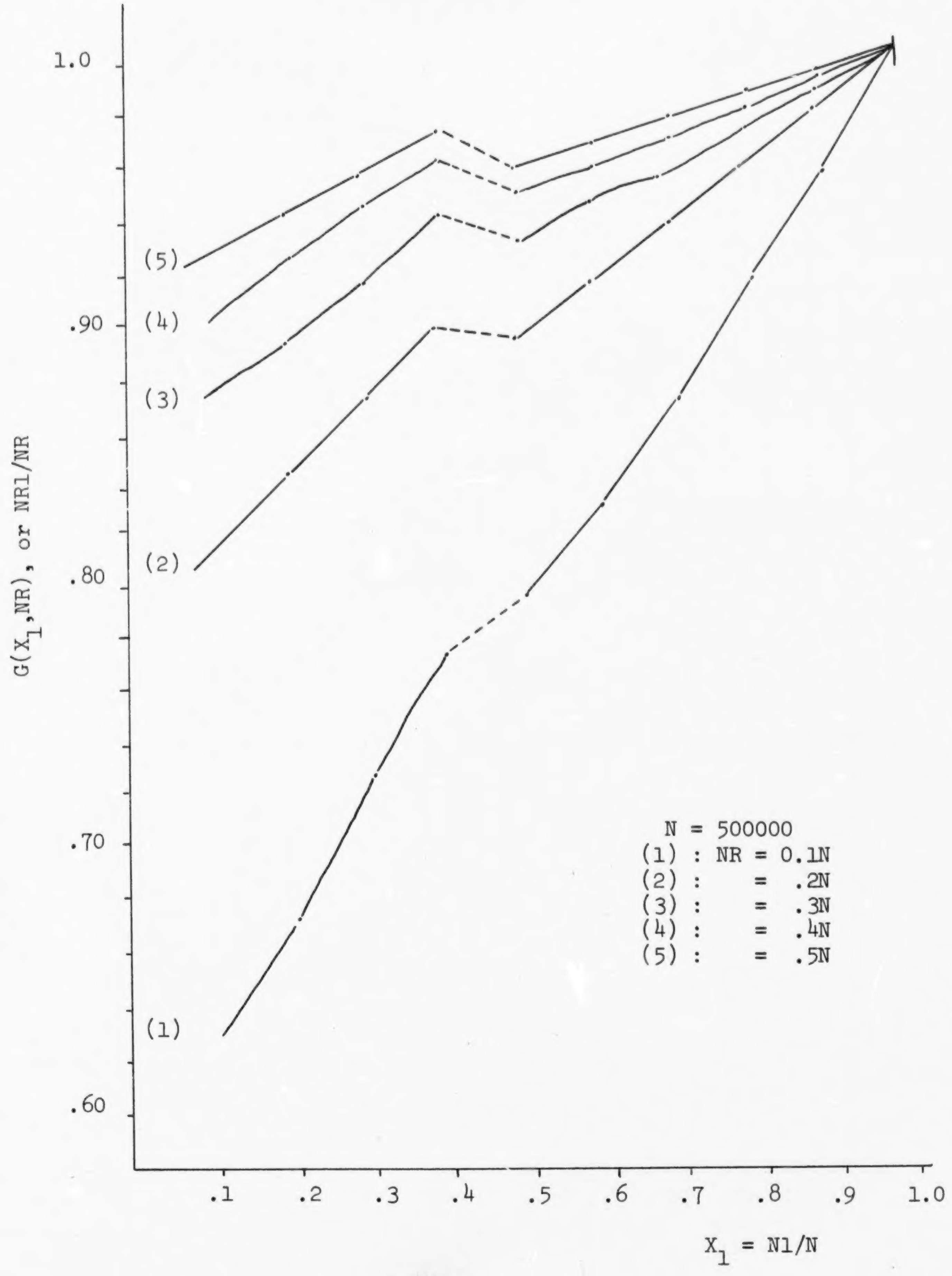


FIG. 47:
CUMULATIVE - REQUIRED DISTRIBUTION FOR A
TWO-DEVICE FILE (2314-2321)

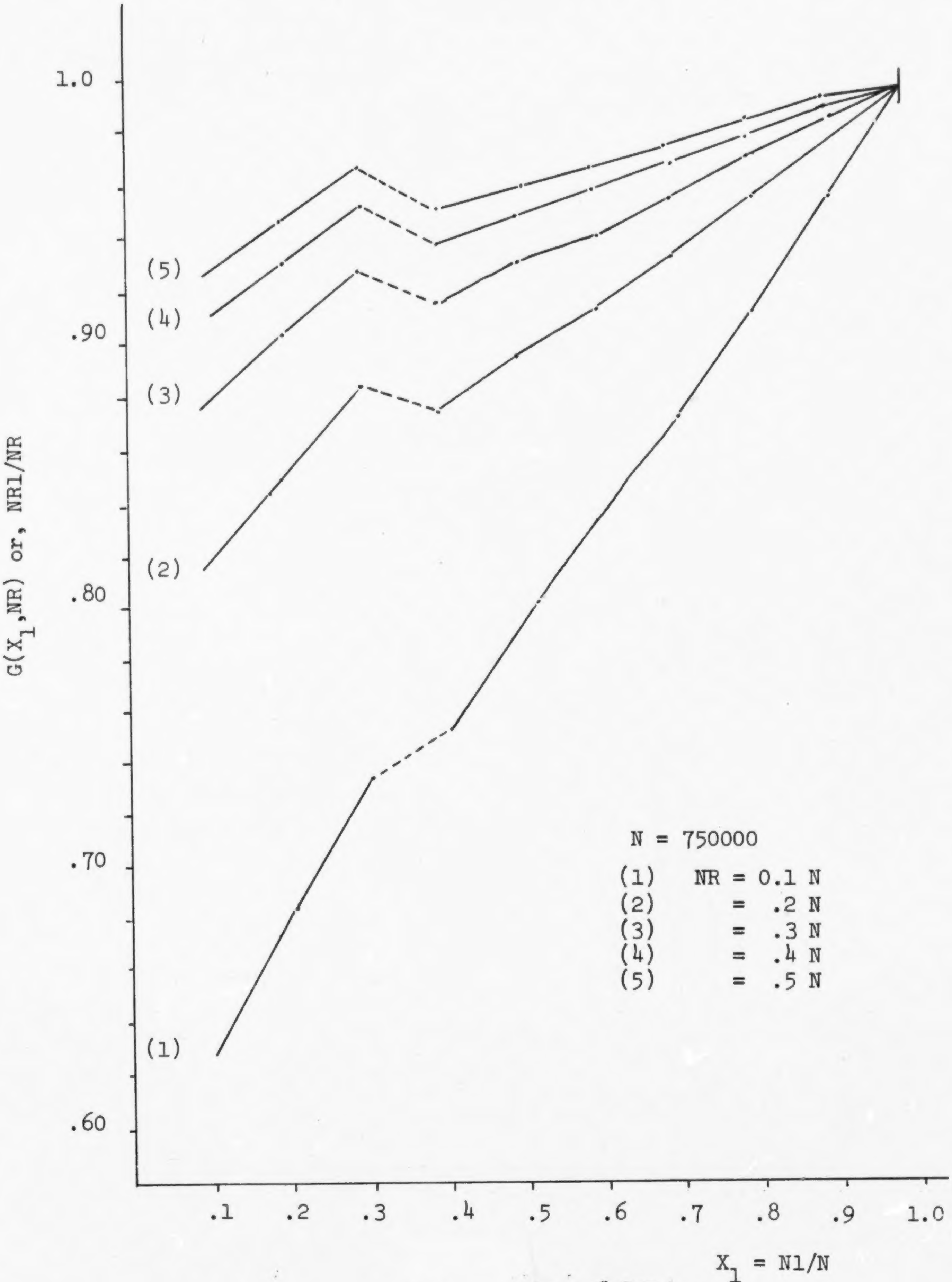
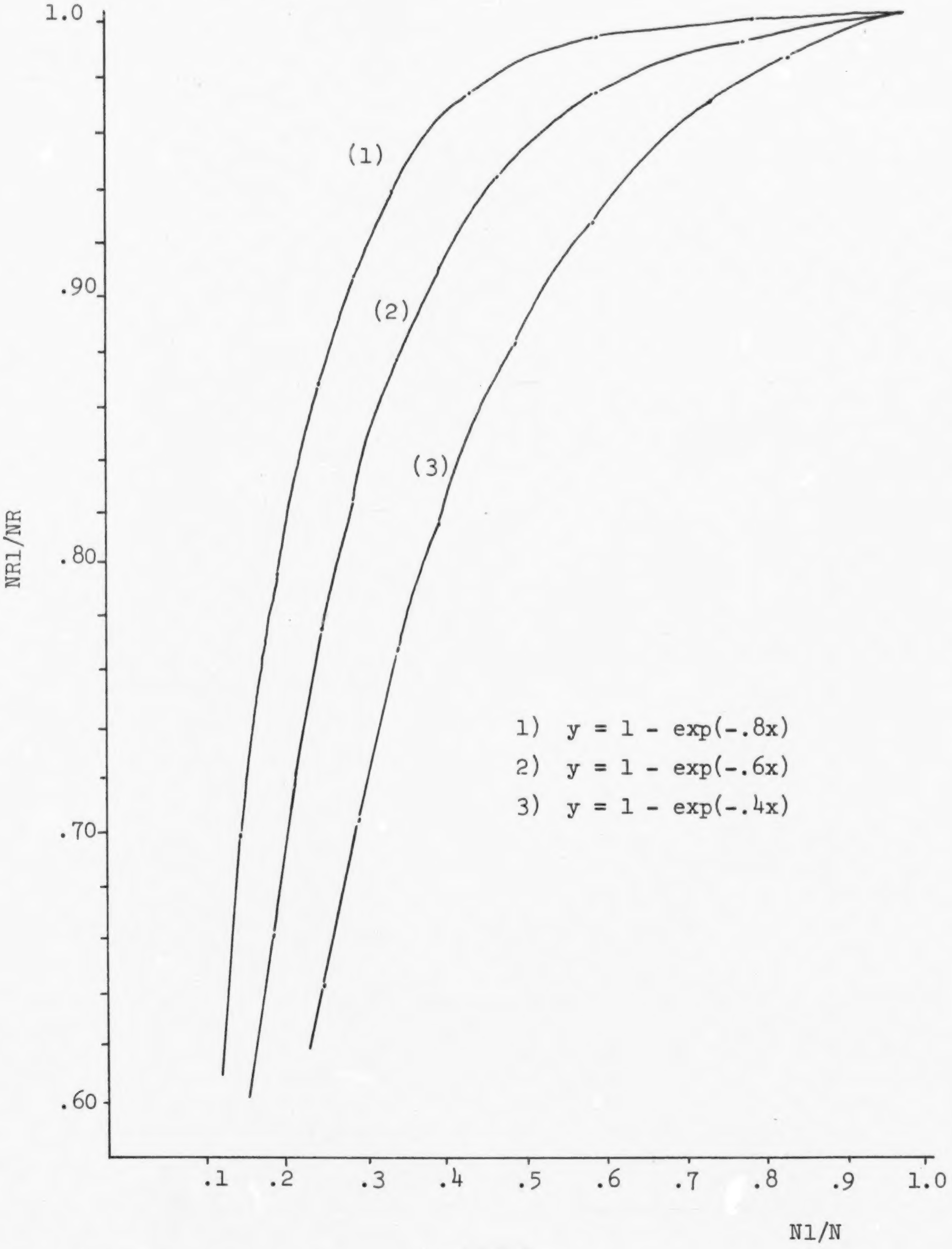


FIG. 48:
ARTIFICIAL CUMULATIVE USAGE
FREQUENCY DISTRIBUTIONS



4.8.4.5 Program DUODEV

Under the assumption that a usage frequency distribution is known, the calculation of optimal division points for two-device files is carried out by program DUODEV. The logic steps followed by this program are:

a. Input the file size and characteristics for the two devices under consideration.

b. Compute a set of ten "cumulative required distributions" (G(X)) and store them in a matrix.

c. Input usage frequency distribution.

d. Compare each of the ten RCD's with the usage frequency distribution, and find the point X_1^* (if it exists) at which $F(X_1) - G(X_1)$ is maximum.

e. Calculate required storage, search time and total search cost per request for the divided file corresponding to each X_1^*.

To test these ideas in advance of the availability of "real-world" data, we used exponential functions to develop "artificial" usage frequency distributions which are plotted in Fig. 48.

4.8.4.6 Some Results

Using program DUODEV, we calculated optimal division points for three different file sizes and a range of ratios NR/N from 0.01 to 0.9. The results are presented in tables 10 and 11, from which one can notice the following:

a. In general, the cost values corresponding to usage frequency #1 are lower than those for usage frequency #2. It is obvious that the larger the parameter of the negative exponential function (in this case .8), the more advantageous the division.

b. For file sizes N=500K and N=750K the division points do not coincide for every NR/N value, and the costs are not equal. On the contrary, we found that the results of the division of a file based on key-lengths were only a function of the ratio NR/N. Figures 45 and 46 may explain this behavior; the set of cumulative required distribution curves changes as file size N does.

It is interesting to compare the results obtained for the division of a file based on key-lengths, and the two-device files based on usage frequency (Table 9). We can see that for usage frequency distribution #1 only the last two costs based on "key-length" division are lower than those based on "usage frequency" division. However, we can not reasonably extend this result to the real case because usage frequency distributions for the system under consideration are presently unknown. We can say, however, that only a very "fast decaying" usage frequency distribution could yield better results than the division based on key lengths. There may exist, of course, other considerations which, when taken into account, would end up giving a more favorable score to the two-device system.

Table 9: Comparison of Search Costs for Two Systems

NR/N	ONE DEVICE		TWO DEVICES	
	I	II	III	IV
0.02	0.0926	0.0649	0.0569	0.0599
0.05	.0399	.0292	.0288	.0309
0.10	.0223	.0172	.0181	.0191
0.20	.01355	.01126	.0118	.0125
0.50	.0083	.00759	.00782	.0082
1.00	.00653	.00631	.00634	.0065

Notes:
I = undivided file
II = one division based on key-lengths
III = two devices for u.f.d. (1)
IV = two devices for u.f.d. (2)

4.8.4.7 An Extension

The following scheme constitutes a natural extension of the vertical segmentation of a file based on usage frequency distribution and employing two devices.

a. The first division is based on usage frequency distribution as before.

b. The first subfile is stored in the faster device, and is searched first, as before.

c. The remaining set of records which do not belong to the first subfile is divided into two subfiles according to key-length distribution.

d. Subfiles 2 and 3 are then stored in a slower device.

Program MIXPART has been devised to handle this case. This is a variation of program DUODEV with the additional use of subroutine OPTWO.

MIXPART calculates the necessary "cumulative required distributions" (CRD), compares them with the given usage frequency distribution (U.F.D.) and finds the first division point. Then, assuming that any subset of records has the same key-length distribution, the program uses subroutine OPTWO to find the second division point. Storage requirements for the three subfiles are calculated, and search time per request is computed based on the assumption that for the second and third subfiles, the volume of requests is proportionate to the subfile size.

Table 10: Total Search Cost as a Function of System Volume for a Two-Device File**

Usage Frequency Distribution 1

NR/N	N = 250K		N = 500K		N = 750K	
	DP*	COST	DP*	COST	DP*	COST
.01	.10	.09757	.10	.09821	.10	.09832
.02	.15	.05636	.15	.05688	.15	.05706
.03	.20	.04126	.20	.04172	.20	.04196
.04	.25	.03325	.25	.03370	.25	.03399
.05	.25	.02832	.25	.02877	.35	.02869
.06	.30	.02485	.30	.02531	.35	.02501
.07	.30	.02236	.30	.02282	.35	.02237
.08	.35	.02047	.30	.02095	.35	.02039
.09	.35	.01893	.35	.01901	.35	.01885
.1	.35	.01769	.35	.01810	.35	.01762
.2	.45	.01194	.50	.01175	.45	.011711
.3	.45	.00989	.55	.0096	.50	.00960
.4	.50	.00883	.55	.00850	.55	.00849
.5	.50	.00818	.55	.00782	.55	.00782
.6	.55	.00774	.60	.00735	.60	.007356
.7			.60	.00701	.60	.00702
.8			.65	.00676	.65	.006764
.9			.65	.00655	.70	.00657

* Division point as percentage of records in the first file.

** First subfile in 2314 DASF, second subfile in 2321 data cell.

Table 11: Total Search Cost per Request as a Function of System Volume for a Two-Device File**

Usage Frequency Distribution 2

NR/N	N = 250K		N=500K		N = 750K	
	DP*	COST	DP*	COST	DP*	COST
.01	.05	.09964	.05	.10055	.05	.10061
.02	.15	.05931	.15	.05995	.15	.06012
.03	.25	.04390	.25	.04445	.25	.04474
.04	.30	.03563	.30	.03617	.35	.03599
.05	.30	.03040	.30	.03094	.35	.03046
.06	.35	.02677	.35	.02732	.35	.02676
.07	.35	.02413	.50	.02460	.40	.02406
.08	.40	.02208	.50	.02229	.40	.02197
.09	.40	.02046	.50	.02050	.45	.02034
.10	.40	.01915	.50	.01906	.45	.01897
.20	.55	.01288	.55	.01250	.55	.01254
.30	.60	.01062	.60	.01018	.65	.01020
.40			.65	.00896	.70	.00896
.50			.70	.008194	.70	.008194
.60			.75	.007674	.75	.00767
.70			.75	.007294	.75	.00729
.80			.80	.00700	.80	.00700
.90			.80	.00677	.80	.00677

* Division points as a percentage of records in first file.

** First subfile in 2314 DASF, second subfile in 2321 data cell.

Some results obtained using UFD #1 are tabulated in Table 12. An overall observation of this table shows that the results obtained are better than those presented in Table 10 and even better than the results for the two divisions based on key-length distribution.

4.8.5 Conclusions

It seems reasonable to state that the order of efficiency for the different structures considered in this study is the following:

1. Two-device system: first division based on usage frequency distribution; second division based on key-length distribution. Devices used: 2314 and 2321 (Program: MIXPART).

2. One-device system using the 2314 storage facility: division based on key-length distribution (Program: DYNPART).

3. Two-device system: Only one division, to be based on usage frequency distribution. Devices: 2314 and 2321 (Program: DUODEV).

However, before generalizing these conclusions, the following remarks should be kept in mind:

-- No parallel operation has been considered.

-- No real usage frequency distribution is known.

As was stated before, these two factors might change the outcome.

It should be noted that both programs DUODEV and MIXPART may be used for a one-device system as well. This is achieved by using as input the same set of data.

If the three basic structures are compared on the basis of search cost per request alone (see Table 13), the two-device, triple segmentation approach (MIXPART) yields the lowest cost for any given value of request volume/file size (NR/N), followed by the two-device, dual segment alternative (DUODEV), and finally the single-device dual segment technique using key length (DYNPART). As request volume rises for a given file size, the multiple-device, triple segmentation approach begins to have payoffs if looked at in terms of gross search cost per month. For example, at NR/N=0.10 the total search cost per month for 50,000 requests, for the structures would be:

MIXPART:	Two-device, 3 segments	$805.50	(.01611 x 50k)
DUODEV:	Two-device, 2 segments	905.00	(.0181 x 50k)
DYNPART:	One-device, 2 segments	957.50	(.01915 x 50k)

This analysis does not include file maintenance costs and other implementation considerations, nor does it consider other device combinations which more likely might be available in a practical sense (e.g. multiple 2314's).

However, we have aimed at laying out a provisional framework to guide designers of large inverted file systems when they know or can estimate the following:

a. size of the master file;

b. number of index records linked to master file and average index record length;

c. average number of master records mapping into an index record;

d. key length distributions of the indexes;

e. system search request volume;

f. usage frequency distributions of indexes and master records;

and either decisions must be made on efficient use of storage within one available storage device, or opportunities for operational minimization by use of available multiple types of devices must be exploited to maximum advantage. And a decision must be reached either to use a single available storage device to best cost advantage, or to try to exploit multiple types of devices to reach the lowest possible cost.

Table 12A: Total Search Cost per Request for a Two-Device File, Based on Usage Frequency Distribution and Key-Length Distribution*

NR/N	N = 250K		N = 500K		N = 750K	
	DP	COST	DP	COST	DP	COST
.01	.15	.08699	.15	.08717	.15	.08734
.02	.25	.05175	.25	.05205	.20	.06229
.03	.30	.03821	.25	.03854	.25	.03884
.04	.30	.03085	.30	.03120	.35	.03089
.05	.35	.02626	.35	.02667	.35	.02611
.06	.35	.02307	.35	.02348	.35	.02293
.07	.40	.02076	.35	.02120	.40	.02056
.08	.40	.01892	.50	.01927	.40	.01872
.09	.40	.01749	.50	.01764	.40	.01729
.10	.40	.01635	.50	.01634	.45	.01612
.20	.50	.01075	.50	.01047	.50	.01047
.30	.55	.00875	.55	.00841	.55	.00841
.40	.55	.00771	.60	.00733	.60	.007329
.50	.60	.00706	.65	.00666	.65	.006662
.60			.65	.[illegible]620	.65	.006198
.70			.65	.00587	.65	.005866
.80			.70	.00560	.70	.005606
.90			.75	.00535	.70	.00540

*First division based on U.F.D. No. 1, and is stored in 2314 DASF. The second subfile is divided according to key lengths. Both second and third subfiles are stored in 2321 data cell.

Table 12B: Comparison of Total Search Cost per Request,
for Three Basic Structures

NR/N	DUODEV 2314 & 2321 DASF Two Subfiles	MIXPART 2314 & 2321 DASF Three Subfiles	DYNPART 2314 DASF Two Subfiles
.01	.09821	.8541	.14276
.02	.05688	.05053	.07408
.03	.04172	.03728	
.04	.03370	.03018	
.05	.02877	.02567	.03288
.06	.02531	.02260	
.07	.02282	.02034	
.08	.02095	.01858	
.09	.01901	.01720	
.10	.01810	.01611	.01915
.20	.01175	.01045	.01222
.30	.0096	.00824	
.40	.0085	.00713	
.50	.00782	.00647	.00799
.60	.00735	.00602	
.70	.00701	.00570	.00715
.80	.00676	.00541	
.90	.00655	.00521	.0066

DUODEV: Division is based on UFD #1 only.

MIXPART: First Division based on UFD #1; Second Division based on Key Length Distribution (KLD).

DYNPART: Division based on KLD only.

N = 500K, for all three programs.

5. FIRST-STAGE MODEL OF THE ECONOMIC EFFECTS OF INCORPORATING A DATA COMPRESSION SYSTEM INTO AN ON-LINE DIRECT-ACCESS STORAGE AND RETRIEVAL SYSTEM

By Kelley L. Cartwright

5.1 Introduction: Compression and Coding

5.1.1 The Advantages and Constraints of Compression

The amount of bibliographic information available in machine-readable form is constantly growing, and can be expected to increase rapidly in the coming years. Eventually, much of this information will be stored in random-access devices, where it will be available in very brief times through on-line consoles. A very significant element of the costs of such systems will be the cost of storing the data in random-access devices. For example, the cost of storing a bibliographic citation on disc is presently about 60 cents per year, so that to store a million of these would cost approximately $600,000 per year. Even a 10% reduction in such costs would represent an appreciable saving. If anything, a reduction of this amount seems conservative, since natural language is highly redundant; thus we would expect bibliographic records to be redundant also. Redundancy means that the number of symbols used to express a given amount of information is usually far greater than the number that would actually be required if the symbols were used with maximum efficiency. Shannon* has developed an expression that allows us to estimate the amount of redundancy in messages. This is the famous entropy formula:

$$H_b = -\sum_{i=1}^{r} P_i \log_b P_i$$

($0<P_i<1$; hence $\log_b P_i<0$; hence the minus sign in the above expression, to give H_b a positive value.)

H_b is the entropy of a message; it is the mean number of individual symbols (out of a total of _b_ available symbols) that would be required to express the same amount of information as does a message in a given set of _r_ symbols, if all redundancy were removed from the message, and given that each symbol of the _r_-length alphabet has a frequency P_i in the message. (The sum of the P_i, of course, is 1.)

The two alphabets may be the same (i.e., $b = r$); thus we could calculate H_{26} in letters for messages in the English alphabet (if we ignored blanks, punctuation, capitalization, all of which could, of course, be included in the alphabet).

For any message and any given value of _b_, H_b decreases as _r_ increases.

---

*Shannon, Claude E. and Warren Weaver, The Mathematical Theory of Communication, Urbana: University of Illinois Press, 1949.

For example, in natural languages H_b decreases as the "alphabet" is changed from one that consists only of single letters to one that includes digrams, trigrams, etc., then words, sentences, and larger bodies of text. This is because the creation of a message in natural language is a Markov process; the occurrence of a given character or string of characters influences the probability of occurrence of succeeding characters. (Example: "Probabilit" may be followed by "y" or "ies", but not, in English, by any other letter or letters. Thus the normal letter frequencies of English do not hold at this point, when the preceding letters are taken into consideration.)*

Before proceeding, it should be noted that we accept as a constraint the assumption that the amount of information in bibliographic citations is properly determined by the catalogers and indexers who create them. Therefore, whatever the information content of a record, this content must not be altered by the storage and retrieval system. The system may, however, alter the internal physical format of the information in any way desired. More succinctly, the system must be able to output exactly what was input. What happens in between is the system's business.

5.1.2 Coding Procedure

The process of converting a string of symbols in an alphabet of _r_ symbols to their equivalent in an alphabet of _b_ symbols is called coding. We are specifically concerned here with converting the natural-language texts of bibliographic records into their equivalents in the binary alphabet of 2 symbols. Shannon's formula tells us that redundancy would be eliminated if each symbol were translated into _H_ bits. Unfortunately, _H_ turns out to be an integer in only very special cases; what is usually done, therefore, is to translate each original symbol into some integral number of bits. In the coding schemes of most computers, this is a fixed number which is the same for all symbols. Greater efficiency can be achieved by assigning a variable number of bits to the original symbols, assigning the smallest number to the most frequently occurring symbol. The mean number of bits per symbol into which the symbols in the original array are translated can then be calculated as follows:

$$L' = \sum_{i=1}^{r} L_i P_i$$

L_i is the length in bits of the binary representation of symbol _i_.
P_i is as above.

---

*As a result, the entropy of English text has been estimated in the range of one to two bits per character. No comparable estimation of the entropy of bibliographic records has been made. For example, see Shannon, Claude E. "Prediction and Entropy of Printed English," _Bell System Technical Journal_, (Jan. 1951), 50-64; and Miller, George A. and Elizabeth A. Friedman, "The Reconstruction of Mutilated English Texts," _Information and Control_ (1, 1957), 38-55.

Unfortunately, it is possible only in special cases (e.g. when $P_1 = P_2 = P_3 = \ldots P_r$, and $r = 2^k$, and k is an integer) to have L' = H; hence we have an illustration of the problem stated by Warren Weaver.*

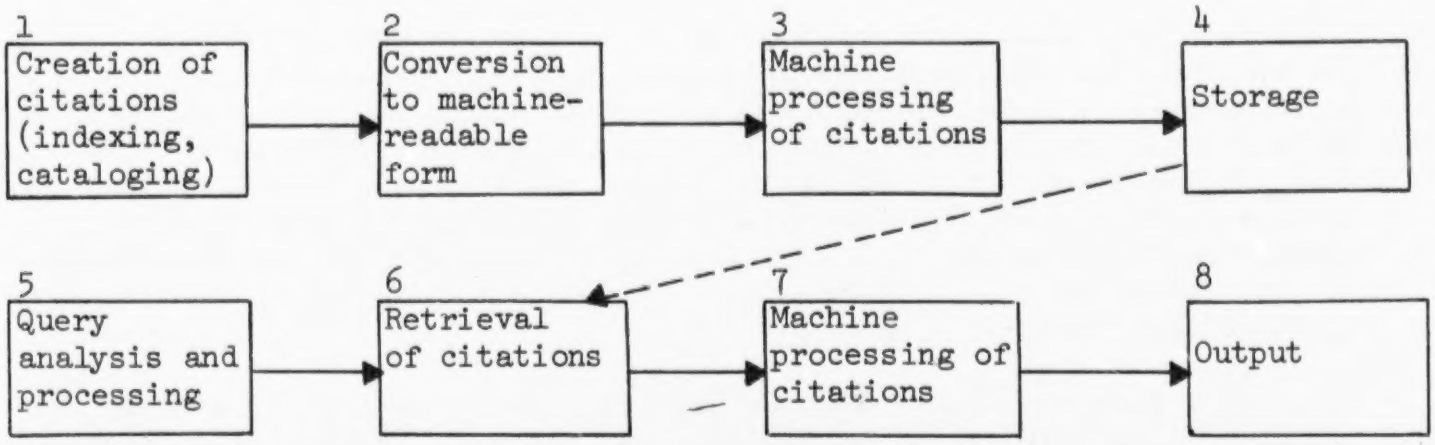
> The statistical nature of *messages* is entirely determined by the character of the source. But the statistical character of the signal as actually transmitted by a channel...is determined both by what one attempts to feed into the channel and by the capabilities of the channel to handle different signal situations. For example, in telegraphy, there have to be spaces between dots and dots, between dots and dashes, and between dashes and dashes, or the dots and dashes would not be recognizable. [Italics Weaver's]

Huffman** has devised a method of variable-length binary encoding which, given all the requirements such a code must meet, results in the minimum possible value of L' for a given series of symbols with known probabilities. It has been decided to use this procedure in the present project.

5.2 A Model of the Effect Upon System Costs of Incorporating a Huffman Coding Scheme into a Storage and Retrieval System

5.2.1 General Effect Upon the System

The following is a very general schematic drawing of the processes involved in mechanized information storage and retrieval:


1
Creation of citations (indexing, cataloging)
2
Conversion to machine-readable form
3
Machine processing of citations
4
Storage
5
Query analysis and processing
6
Retrieval of citations
7
Machine processing of citations
8
Output

Steps 1, 2, 5, and 8 will be the same regardless of whether compression is used. (It is assumed that in Step 5 indexes are consulted--by the computer--and that this process yields the disc addresses of a number of records. Then, in Step 6, these are found and transferred to the CPU.)

---

* Shannon and Weaver, *op. cit.*, p. 108.

**Huffman, D.A., "A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE*, (40, Sept. 1952), 1098-1101. Also in Willis Jackson, Ed., Communication Theory, New York: Academic Press, 1953.

If some or all of the citations stored in Step 4 are stored in a compressed (Huffman-encoded) form, then the following other steps will be affected as indicated:

Step 3. This must now include the compression routines; hence its cost will be increased.

Step 6. This step will be decreased in cost for all compressed records, since the shorter a record the less time it takes to transfer it from disc to the CPU.

Step 7. Cost of this step will be increased because each compressed record will have to be "decompressed" prior to output.

5.2.2 General Evaluation Criteria

In general terms, the criteria for deciding whether to incorporate code compression into a storage and retrieval system are two. First, the cost of the system which includes compression must be less than the cost of the same system without it; and second, the effectiveness of the system must not be reduced significantly.

It appears possible that effectiveness could be reduced in two ways:

a. Turn-around time. The expansion routine will increase the time between the user's submission of his request and his receipt of results. It is assumed, however, that this increase will be measurable in terms of microseconds, or at most, milliseconds, and will be imperceptible to the user.

b. Error rate. An error is an unintended non-correspondence between the data as input and as output, and--for present purposes--attributable to the fact that a compression system has been incorporated. We know that computers introduce a certain level of random error into data, by occasionally altering one or more bits. In a data-handling system in which the data are encoded in a fixed-length scheme, this means that occasionally a character will be altered; but the redundancy in the record will usually mean that this alteration does not seriously distort the message.

In a variable-length scheme, however, the alteration of even a single bit may have more serious effects, altering successive characters. However, for the purposes of this paper, it is assumed that this error level will be low enough to be tolerable, as it usually is in systems not involving compression.*

5.2.3 General Model

The attempt of this paper is to arrive at a model that reflects the difference in cost per stated period of two storage and retrieval systems that are identical in every way except that one incorporates a data compression system and the other does not. Let us call the difference in the cost of these two systems D; and, since it is expected that the difference will favor the system incorporating compression, we will need a model in the form

---

*Extensive work has been performed on limiting the error propagation of variable-length codes. See for example: Neumann, Peter G., "Efficient Error-Limiting Variable-Length Codes," IRE Trans. on Information Theory, (July, 1962).

$$D = U - C$$

This equation introduces a convention on variable naming which is used in the balance of this paper. Cost components will be named in terms of the steps in the general schematic diagram (Section 5.2.1) in which they are incurred. Thus the variable S3 will represent the costs incurred in Step 3; specifically, it will represent those costs in Step 3 which vary depending upon whether the system incorporates data compression. When we discuss Step 3, this step will be broken down into steps (a) through (g), and it will be asserted that only steps (e), (f), and (g) will vary depending upon whether the system incorporates data compression; hence, we will discuss the variables S3e, S3f, and S3g. S3f, it will be shown, has three components; hence, we will discuss the variables S3f1, S3f2, and S3f3. It should be clear, then, that any variable beginning with "S" represents the sum of all longer variables beginning with the same characters. Thus, for example, S3 = S3e + S3f + S3g, and S3f = S3f1 + S3f2 + S3f3.

Further, any variable name that begins with S refers to the cost of a step without regard to whether the system incorporates data compression. If we are discussing one of the variables as it applies in a system with data compression, then I will substitute "C" for "S" in the variable name; if we are discussing an "S" variable as it applies in a system without data compression, then I will substitute "U" for "S". Thus "S4a" refers to the cost, in either system, of transferring a record to disc; "C4a" refers to this cost in a system with compression; "U4a" refers to this cost in a system without compression.

On this basis, it can be seen that the above equation "D = U - C" is a simplification of the following:

$$D = (U3 + U4 + U6 + U7) - (C3 + C4 + C6 + C7)$$

I regret imposing variable names like "S3f2" upon the reader, but this procedure saves much complex subscripting and has the advantage that the name gives a clue to where it is discussed. And these will be dropped in the final composite model. There are other variable names not relating to specific steps which are listed in Fig. 49, Index of Variable Names.

5.2.4 Analysis of Variables

5.2.4.1 Step 3: Machine Processing Prior to Storage

The input to this step will be a citation in machine-readable form; the output will be a citation ready for storage on disc. The processing steps include principally (a) machine editing, (b) conversion of the input format to the prescribed storage format, (c) creation of new index entries and updating of existing ones, (d) assignment of disc addresses to each new record, (e) determination of whether the record is to be compressed, (f) compression of those records that are to receive it, and (g) movement of the record to an output area from which it can be transferred to disc.

The steps in this process which will vary depending upon whether compression is incorporated are (e), (f), and (g). These will now be considered in turn.

Step 3(e). The process of examining a record to determine whether it is to be compressed should be a straightforward one, involving logical operations upon such characteristics of each citation as its date of publication, language, and subject area. When statistics on these characteristics are available, and the algorithm has been developed, a mean cost per record for this step can be determined.

Cost component of Step 3(e):

S3e: mean cost per period of determining for every citation newly entered in the system whether it is to be compressed.

Methods of determining values:

U3e	0
C3e	dR

Step 3(f). An Encoding Dictionary will be used to convert natural language records into Huffman-encoded records. It is assumed that this dictionary will itself be stored on disc and brought into main memory whenever encoding is to take place. It is further assumed that this transfer process involves moving the dictionary to an input area and then to a work area. (Note that it may be desirable to store the dictionary on tape, depending upon how often encoding takes place.)

The average cost of applying the encoding algorithm to a record is the sum of the number of encodable single characters times the cost of applying the algorithm to a single character, plus the number of encodable digrams times the cost of encoding a digram, etc. An "encodable" symbol is one for which a Huffman code has been provided. Mean number of encodable symbols of lengths 1, 2, ..., n per record will have been estimated carefully at the time that the encoding dictionary was established, and the amount of time required to encode a symbol of length $\underline{i}$ can be predicted after the encoding algorithm has been flowcharted.

FIG. 49:
INDEX OF VARIABLE NAMES

	Variables representing the cost of applying algorithms
a	The cost of checking a record retrieved from disc to determine whether it is in compressed or uncompressed form.
b	Mean cost per record to be compressed, of applying the encoding algorithm.
c	Mean cost per application of applying the decompression algorithm to a compressed record.
d	Mean cost per record of analyzing a record to determine whether it should be compressed.
	Variables representing equipment costs
q	Cost of maintaining a byte on disc for one period. This is an equipment cost only; it is simply the rental (or depreciated purchase) cost of the disc and its interface equipment per period, divided by the number of bytes that can actually be stored on the disc. (The latter number is usually somewhat less than the disc's capacity.)
r	Cost of transferring a byte from disc to an input area of core storage, or from an output area of core storage to disc.
t	Cost of positioning the read/write mechanism of a disc drive at a specified address on the disc.
u	Cost of moving one byte from one position in storage to another.
	Variables representing system characteristics
A	Mean length (in bytes) of uncompressed records in storage format.
B	Mean percentage reduction of record size achieved through data compression.

FIG. 49 (Cont.)

Variables representing system characteristics (cont.)	
E	Number of times per period that the encoding dictionary is transferred from disc. If input is done on line, then it may be desirable to encode each record as it is input, if it is to be encoded. In this case, we will have $E = PR$.
F	Length (in bytes) of the encoding dictionary.
G	Number of times per period that the decoding dictionary is brought into main storage. If this is done each time a record is to be decompressed, then $G = Y$; or if it is done, for example, *n* times per working day, then $G = n$ times the number of working days per period.
H	Length (in bytes) of the decoding dictionary.
P	The percentage of R which receive compression ($0 < P < 1$; $P = 1$ only if *no* records are stored in uncompressed form).
R	The number of citations added to the file per period.
X	The number of uncompressed records retrieved from disc per period.
Y	The number of compressed records retrieved from disc per period.

Cost components of Step 3(f):

S3f1: Cost per period of storing the encoding dictionary.
S3f2: Cost per period of bringing the encoding dictionary into core storage each time it is needed.
S3f3: Cost per period of applying the encoding algorithm.
S3f4: Cost per period of moving new records from a work area in storage to an output area.

Methods of determining values:

U3f1	0
U3f2	0
U3f3	0
U3f4	RAu
C3f1	qF
C3f2	$E(t + rF + uF)$
C3f3	bPR
C3f4	$u(RA[1 - P] + RAP[1 - B])$

5.2.4.2 Step 4: Storage on Disc

Two cost components involved here are (a) the transfer of records from the output area of the CPU to the disc, and (b) their storage on the disc during the stated period. It is assumed that the new records will enter the system at a constant rate during the period; hence, the average time that records will be resident on the disc during their first period of inclusion in the system will be one-half period.

Cost components to Step 4:

S4a: Cost per period of transferring newly entered records from core storage to disc.

S4b: Cost in the nth period of maintaining on disc the records entered during the 1st through nth periods.

Methods of determining values:

U4a RAr

U4b $q\left[\sum_{i=1}^{n-1} R_iA + .5R_nA\right]$

C4a $RAr(1 - P) + RArP(1 - B)$

C4B $q\left[\sum_{i=1}^{n-1} [R_iA(1 - P) + PR_iA(1 - B)] + .5[R_nA(1 - P) + PR_nA(1 - B)]\right]$

Note that the above assumes the values of P and B remain constant over time. Actually, the expression for C4b will need to be modified because some records originally stored in uncompressed form will later be stored in compressed form. Also, t is not a variable in U4a and C4a because

R will not vary depending upon whether compression is used. Hence tR will be the cost in both systems per period of finding the specified addresses of records ready to be stored on disc.

5.2.4.3. Amortization of Costs of Steps 3 and 4 (a)

The costs of S3e, S3f, S3g, and S4a are all one-time costs whose benefits will be realized over time; it seems reasonable, therefore, to amortize them over a number of periods. These four cost components may be collectively characterized as "initial processing costs." Let us assume that it has been decided to amortize these costs over m periods, and that we are in the nth period. Then we will assign to each period, in place of the costs S3e, S3f, S3g, and S4a, a cost equal to the following sum:

$$\frac{1}{m} \sum_{i=j}^{n} (S3_i + S4a_i)$$

Note: $j = n - m + 1$

(Actually, S3f1 will be a constant value from period to period, so that we could more properly write a more complex expression to take account of this fact.)

5.2.4.4 Step 6: Retrieval of Citations

The input to this step is the disc address of a record. At the end of it we will have in the input area of main memory a record in exactly the form in which it was stored on disc. The steps involved are (a) go to the specified address on the disc; and (b) transfer the record found at that address to main memory. The only cost component that will vary depending upon whether data compression is employed is Step 6(b).

Cost component:

S6b: Cost per period of transferring retrieved records from disc to core storage.

Method of calculating values:

U6b	XAr
C6b	XAr + YA(1 - B)

5.2.4.5 Step 7: Processing of Citations for Output

The input to this step is a record (compressed or uncompressed) located in the input area; the output must be a record ready to be displayed on an output device. The major steps are the following: (a) the record is moved from the input area to a work area; (b) the record is checked to determine whether it is compressed; (c) if it is compressed, it is decompressed; (d) the record is converted from storage format to output format; (e) the record is moved to the output area. Steps (a), (b), and (c) are the steps that will vary depending upon whether the system incorporates data compression. Step 7(b) will be a very straightforward operation, probably involving no more than checking the first bit in the record. Step 7(c) will take place in two steps--(1) a decoding dictionary must be brought in from disc; and (2) the record must be decoded. Thus we are confronted here, as in Step 3(f), with a continuing dictionary storage cost. We are also confronted, as in several other areas, with the cost of applying an algorithm to a record--a cost that can be estimated carefully only on the basis of detailed flowcharts that have not yet been developed.

Cost components:

S7a: Cost per period to move retrieved records from the input area to the work area.

S7b: Cost per period of checking records retrieved from disc to determine whether they are compressed or uncompressed.

S7c1: Cost per period of storing the decoding dictionary on disc.

S7c2: Cost per period of transferring the decoding dictionary from disc to core storage.

S7c3: Cost per period of decoding retrieved compressed records.

Methods of determining values:

U7a	uAX
U7b	0
U7c1	0
U7c2	0
U7c3	0
C7a	$u(AX + AY[1 - B])$
C7b	$a(X + Y)$
C7c1	qH
C7c2	$G(t + rH + uH)$
C7c3	cY

5.2.5 The Detailed Model

Fig. 50 presents the combined values of U and C, and their combination, via the general model "D = U - C," into a formula for D. The formula has the form one would expect - an expression representing the savings in record storage and transmisison costs minus a number of costs representing the storage of the encoding and decoding dictionaries and the compression and decompression routines.

FIG. 50:
COMBINED MODEL

GENERAL MODEL: $D = U - C$

$$\begin{aligned} C = {} & A[q(1 - PB)(N + .5R_n) + (u + r)(Y - YB)] \\ & + (1/m)M[d + bP + A(u + r)(1 - PB)] \\ & + E[t + F(u + r)] + G[t + H(u + r)] \\ & + q(F + H) + cY + a(X + Y) \end{aligned}$$

$$U = A[(1/m)(u + r)M + q(N + .5R_n) + (u + r)X]$$

$$N = \sum_{i=n-m+1}^{n-1} R_i \qquad M = \sum_{i=1}^{n} R_i$$

m = the number of periods over which initial processing costs are amortized.

$$\begin{aligned} D = {} & A[PBq(N + .5R_n) + (X - Y - YB)(u + r)] \\ & - (1/m)M[d + bP + A(u + r)(1 - PB)] \\ & - E[t + F(u + r)] - G[t + H(u + r)] \\ & - q(F + H) - cY - a(X + Y) \end{aligned}$$

The most important use of this model is as an aid in determining whether to incorporate a data compression system into a storage and retrieval system. In particular, one wants to estimate the effect of the values of P and B upon the value of C, and consequently upon the value of D. The reason that it is assumed that $P < 1$ is that there would appear to be some number n such that if the number of times that a given record is retrieved during a given period is greater than or equal to n, then the total cost of compressing the record, storing it in compressed form, and decompressing it each time it is retrieved will exceed the cost of simply storing it in uncompressed form. An exact model of this relationship needs to be worked out.

The latter model will allow us to predict the value of P, with the characteristics of the records known. This is clearly a crucial variable in the expression shown for D in Fig. 50. More crucial is the value of B, which itself partially determines the value of P as well as affecting the values of b, c, F, H, X, and Y. Hence the most important next step is to determine how these vary in relation to B, which must be done on the basis of data not yet available.

5.3 Huffman Codes

A method of constructing compact binary codes was developed by D.A. Huffman.* The method is the following. A source alphabet of n symbols has associated with it a set of n symbol probabilities $(P_1, P_2, P_3, \ldots P_n)$.

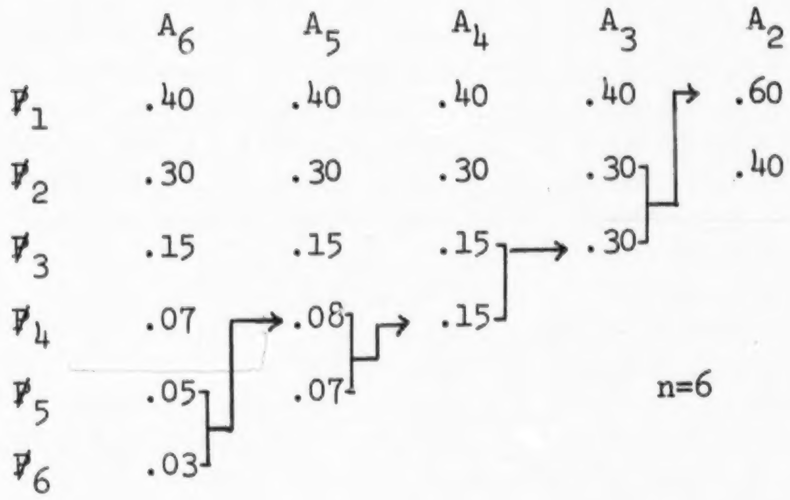
$$[0 < P_i < 1, \text{ and } \sum_{i=1}^{n} P_i = 1]$$

First rearrange and renumber these probabilities so that $P_1 \geq P_2 \geq P_3 \geq \ldots \geq P_n$. Form the sum of the two smallest probabilities $(P_n + P_{n-1})$. We now have a set of (n-1) probabilities. Rearrange and renumber the set so that $P_1 \geq P_2 \geq P_3 \geq \ldots \geq P_{n-1}$. Again add the two smallest probabilities $(P_{n-2} + P_{n-1})$ to form a set of (n-2) probabilities. Continue this process, each time forming an ordered set of probabilities that is one probability shorter than the preceding set. Let us call these sets of probabilities A_n, A_{n-1}, A_{n-2}, etc. When the set A_2 is formed (by reducing A_3), the process stops. Fig. 51 illustrates the process.

---

*Huffman, D.A., *op. cit.*

FIG. 51:
REDUCTION OF A SET OF PROBABILITIES



Note that in any set A_i (except A_n), one of the probabilities is the sum of two of the probabilities of A_{i+1}.

The first step in constructing a code on the basis of this reduction process is to assign the codes 0 and 1 to $\not P_1$ and $\not P_2$ in A_2. Now let $\not P_j$ be the probability in A_2 which is the sum of $\not P_2$ and $\not P_3$ of A_3. The codes for $\not P_2$ and $\not P_3$ of A_3 are formed by putting a one and a zero on the end of the code for $\not P_j$. Thus if the code for $\not P_j$ is "1", the codes for $\not P_2$ and $\not P_3$ of A_3 are "10" and "11".

This process is now repeated to form codes for $A_4, A_5, \ldots, A_n$. In each case the code associated with the probability $\not P_j$ of A_i which is the sum of the probabilities $\not P_i$ and $\not P_{i-1}$ of A_{i+1} is converted to the codes for these two probabilities by putting a "0" and a "1" on the end of the code associated with $\not P_j$ and A_i. This process is illustrated in Fig. 52.

FIG. 52:
CREATION OF A COMPACT CODE

A_2	Code	A_3	Code	A_4	Code	A_5	Code	A_6	Code
.60	*0	.40	1	.40	1	.40	1	.40	1
.40	1	.30	00	.30	00	.30	00	.30	00
		.30	*01	.15	010	.15	010	.15	010
				.15	*011	.08	*0110	.07	0111
						.07	0111	.05	01100
								.03	01101

Based on Fig. 51.
Codes marked with an asterisk (*) are those which, at the next stage, will be decomposed (by having a "0" and "1" added to them) to produce two new codes.

The code shown in the rightmost column of Fig. 52 satisfies all of the criteria we have established for an optimal binary code. Its mean symbol length is the following:

$$\bar{L} = \sum_{i=1}^{n} P_i L_i$$

$$= .4+(2x.3)+(3x.15)+(4x.07)+(5x.05)+(5x.03)$$

$$= 2.13 \text{ bits per symbol}$$

This code satisfies all the criteria we have established. It is not the only code which will do so, however. For example, the following are acceptable alternatives to the above code:

0	1	0
10	01	11
110	001	100
1110	0001	1010
11110	00001	10110
11111	00000	10111

These codes are equivalent and any might be determined by the Huffman method; their lengths satisfy the criterion of compactness. We may therefore use a somewhat simplified version (Fig. 53) of the procedures shown in Fig. 52. Now when we retrace our steps through the reduction process we do not actually synthesize codes, but calculate their lengths. From Fig. 53 we see that a compact variable-length code for a 6-character alphabet having these symbol probabilities will have a 1-bit code for the most frequent character, a 2-bit code for the second most frequent, etc.

FIG. 53: EFFECT OF CHARACTER FREQUENCY ON NUMBER OF BITS IN COMPACT VARIABLE-LENGTH CODE FOR 6-CHARACTER ALPHABET

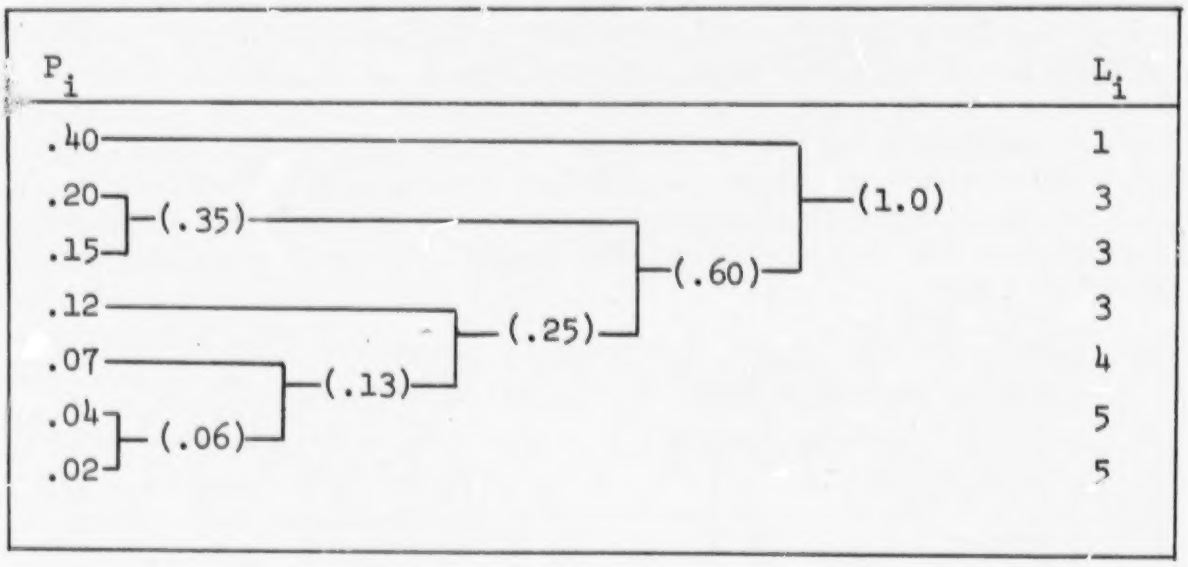
A_2	L_i	A_3	L_i	A_4	L_i	A_5	L_i	A_6	L_i
*.60	1	.40	1	.40	1	.40	1	.40	1
.40	1	.30	2	.30	2	.30	2	.30	2
		*.30	2	.15	3	.15	3	.15	3
				*.15	3	*.08	4	.07	4
						.07	4	.05	5
								.03	5

This example may tempt the reader to suppose that given n probabilities, with $\not{P}_1 \geq \not{P}_2 \geq \ldots \geq \not{P}_n$, the corresponding code lengths will be 1, 2,...,n-1, n-1 bits. This is not the case. Nor is it true that the Huffman code for the most frequent source symbol will have length one. The characteristics of a Huffman code we can predict, given $\not{P}_1 \geq \not{P}_2 \geq \ldots \geq \not{P}_n$, are that $L_i \leq L_2 \leq \ldots \leq L_{n-2} \leq L_{n-1} = L_n$.

Another method of computing symbol lengths for Huffman codes derives from the rule that the length of the code associated with each probability in the original set will be one greater than the number of times that it or another probability which is a sum that includes it was combined to form a sum. Thus, in Fig. 51, $\not{P}_4$ (.15) was first used to form $\not{P}_3$ (.30) in A_3; this .30 was used to form $\not{P}_1$ (.60) in A_2. Hence L_4 should be 1+2=3; this is indeed the length shown in Fig. 53.

The latter fact leads to the idea of using a tree structure to compute code lengths. The probabilities are arranged in descending order, as before, and the tree structure created by repeatedly joining together the two least probabilities that have not already been joined. The process is shown in Fig. 54. The lengths of the Huffman codes are determined by tracing a path through the tree from 1.0 to each original probability. The number of branches one is required to take is the length of the Huffman code for the probability in question.

FIG. 54:
TREE STRUCTURE FOR COMPUTING HUFFMAN CODE LENGTHS



6. IMPLEMENTATION OF BIBLIOGRAPHIC RECORD COMPRESSION

By Vikas Sahasrabudhe and Ashok Kulkarni

6.1 Overview

The first step in exploring the general model of the effects of bibliographic compression is to develop a set of programs for analyzing the records, and for encoding them into the compressed representation and for decoding them to restore the original representation when processing or display is required. Initial exploratory work was carried out which led to the development of programs for the CDC 6400 (written in the assembly language COMPASS). Following the brief results obtained from the LC MARC test tape, a modified design was established which was coded in assembly language for the IBM 360 so that it could be incorporated in the ILR file organization system, CIMARON.

These programs have parameter controls which allow varying degrees of compression to be obtained through control of the size and composition of the encoding "alphabet". However, additional study and analysis has revealed some errors in the design which need to be corrected. These modifications are identified in the report. As a result of the changes needed, these routines have not been incorporated into CIMARON. Finally, the results obtained thus far indicate that there is indeed practical value to be derived from the incorporation of compression techniques as part of on-line bibliographic systems.

6.2 Record Encoding

The discussion of encoding and decoding presupposes that an "alphabet" of source symbols, consisting of single characters, digrams, trigrams, and longer polygrams each will have individual Huffman codes assigned to them. The encoding program will scan the record from left to right, and at each successive point in this scanning will encode the longest possible string with a single Huffman code. It will then proceed to the first character beyond this string and repeat the process.

For example, suppose that the system is being operated on a machine with a 6-bit byte, so that 2^6, or 64, individual characters must be included in the source symbol alphabet. Let us call these c_1, c_2, ..., c_{64}. Suppose further that the digrams c_1c_2 and c_2c_3 have been included in the alphabet, as has the trigram $c_1c_2c_3$. The digram c_1c_1 and the quadrigram $c_1c_2c_3c_1$ have not been included. Then the message $c_1c_1c_2c_3c_1c_2$ would receive 3 Huffman codes - the first for c_1 alone, the second for $c_1c_2c_3$, and the third for c_1c_2.

Three major problems arise in implementing such a scheme. The first is that of devising an encoding algorithm, the second is that of devising a decoding algorithm, and the third is the determination of the series of symbols of which the "alphabet" is to be composed. The first two problems are dealt with in this section, and the third in Section 6.5.

6.3 Table Structure

The encoding algorithm is based upon the use of a dictionary. A method of dictionary construction and use developed by Lamb *et al.** for use in mechanical translation work has been adapted for the present purpose. This dictionary consists of a "first-letter table," which will have a number position corresponding to each character in the computer's particular character set which is also a member of the source alphabet. These are backed up by "second-letter-tables," "third-letter-tables," etc., depending upon the maximum length of a polygram included in the source alphabet.

6.3.1 First Letter Tables

Each position in this table which corresponds to a source alphabet character will contain one of two kinds of entry--either a "termination entry" or a "continuation entry." The former will be present if no programs beginning with the character in question have been included in the source symbol alphabet; otherwise, the continuation entry will be present. Both forms of entry will begin with a single bit that indicates their type - e.g., a "0" may indicate a continuation entry and a "1" a termination entry.

Following the bit which indicates entry type will be a string of $N+1$ bits, where N is the number of bits in the longest Huffman code that must be stored in the first-letter table. A single extra bit (the "1" in the above sum) will be a *demarcating* bit. Its function will be clear after the complete structure of the termination entry is given in detail. For that purpose let us let $N=19$; then each entry in the first-letter table must be 19 bits + 1 indicater bit + 1 demarcating bit = 21 bits long. Let us assume that in the computer in question it is reasonable to make each entry 24 bits long (3 bytes on the IBM 360, 4 on the CDC 6400). The last three bits will be filled with anything (let us say 0's for illustration). Let us also assume that the demarcating bit will be a "1". Finally, let n be the length in bits of an individual Huffman code. n will range in value from some minimum to N. Then an entry in the first-letter table will have the following structure:

Length	1 bit	m bits	1 bit	n bits	3 bits
Contents	1	00...00	1	xx...xx	000
Description	Indicator bit	Filler	Demarcating bit	Huffman code	Filler

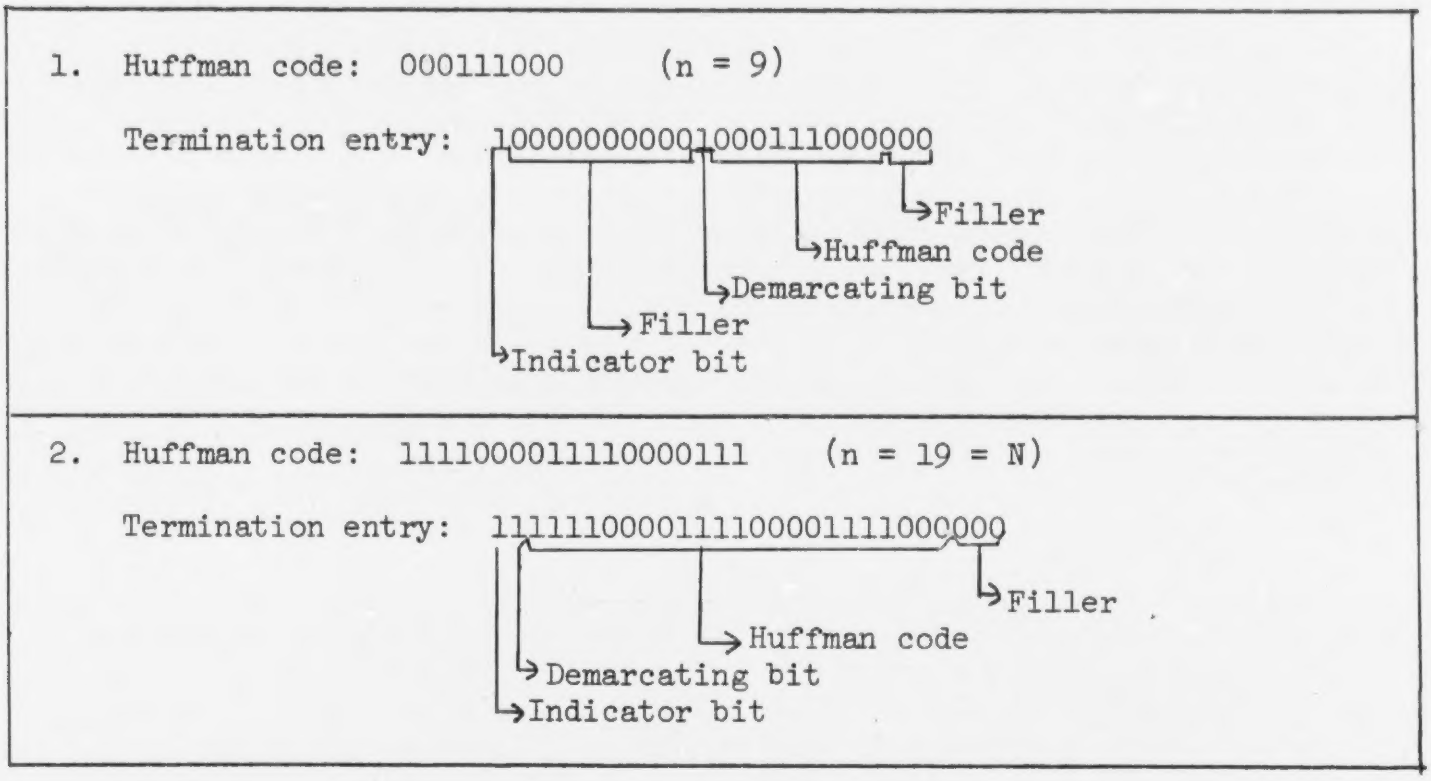
m in the above illustration is a number so chosen that $m + n + 2 = N + 2$.

Figure 55 presents two specific examples of termination entries for the illustrations Huffman codes. The second of these illustrates that if m = N, then m = 0.

---

*Lamb, Sydney M. and W. H. Jacobsen, Jr., "A High-Speed Large-Capacity Dictionary System," *Mechanical Translation*, (6, 1961), 76-107.

FIG. 55:
EXAMPLES OF TERMINATION ENTRIES FOR ILLUSTRATIVE HUFFMAN CODES


1. Huffman code: 000111000 (n = 9)
Termination entry: 100000000001000111000000
Filler
Huffman code
Demarcating bit
Filler
Indicator bit
2. Huffman code: 1111000011110000111 (n = 19 = N)
Termination entry: 111111000011110000111000000
Filler
Huffman code
Demarcating bit
Indicator bit

The algorithm to extract the Huffman code from these entries would scan the termination entry from left to right (after determining that the first bit is a "1"), and the next "1" encountered would indicate that the balance of the entry (except the last three bits) is the Huffman code. It would then be inserted into the coded record.

(The demarcating "1" is not, of course, a part of the Huffman code. The demarcating bit would be unneccessary if the code were so designed that all Huffman codes entered in the first-letter table actually began with the same bit.)

Every character that is the initial character of one or more polygrams that are included in the source "alphabet" will have a continuation entry in the first-letter table. Each such entry will have two parts - (1) the indicator bit that identifies the entry as of the continuation type, and (2) the address of the first position of the second-letter table which contains entries for characters that may be the second character of polygrams that are included in the symbol alphabet and which begin with the character corresponding to the position in the first-letter table in which the continuation entry was found. Note that if the indicator bit is the leftmost bit in the entry, and is a zero, and if the rest of the entry is simply the binary representation of the proper address, then the entry can be used directly for addressing, without the necessity of manipulating it in any way.

Note that the length of each entry in the first-letter table will be determined by the length of the longest Huffman code that must be stored in it, or by the number of bits required to store the address of the second-letter table with the highest address - whichever is longer.

6.3.2 Second-Letter Tables

The structure and use of the second-letter tables can best be made clear with an example. Suppose that a separate Huffman code has been provided for the digram "de" (and, for the moment, suppose that no longer polygrams beginning with "de" have been given separate Huffman codes). Then, when a "d" is encountered in a bibliographic record, the continuation entry in the first-letter-table position corresponding to "d" would give the address of the second-letter table corresponding to "d". We would now inspect the next character in the bibliographic record and, if it is an "e", we would go to the "e" position in the "d" second-letter table. Here we would find a termination entry giving us the Huffman code for the digram "de." (Alternatively, if we had provided Huffman codes for some longer polygrams beginning with "de," we would find a continuation entry.)

A not insignificant problem is imbedded in the above description. There it is specified that we must "go to the 'e' position in the 'd' second-letter table." The problem is how to determine the address of this "e" position. One way would be to calculate it on the basis of the numeric value of "e" in the computer's fixed-length coding scheme. This, however, could lead to very long second-letter tables, many positions of which would not be used. Each such table would have to be as long as the highest numeric value of a character included in that table, times the length of each position in the table.

Another method that would result in considerably smaller tables would involve translating the code for each second letter to a value equal to its position in the second-letter table. That is, when a continuation entry is found in the first-letter table, the code for the next character in the record would be converted to a value which, when added to the address found in the first-letter table, would be the address of the desired position of the second-letter tables. The values to which the codes were converted would be such that there would be no empty positions in the table caused by gaps between the numeric values of the characters.

There will be a second-letter table for each character in the first-letter table which begins one or more polygrams for which Huffman codes have been provided. The suggested method of addressing these tables suggests that they should be of the same length, and that the nth position in each such table should correspond to the same second character. Thus each position in a second-letter table will refer to a unique digram--the one composed of the character whose entry in the first-letter table contained the base address of the second-letter table in question, plus the character which corresponds to a specific position in that second-letter table.

Let us assume that the code-conversion method of addressing the second-letter tables is adopted, that all second-letter tables are of the same length. Let c_i be a character for which a second-letter table has been provided and let c_j be one of the characters for which an entry is provided in each second-letter table. Then when the string c_ic_j is encountered in a record, three conditions may apply:

1. A Huffman code has been created for c_ic_j, but no Huffman codes have been assigned to polygrams of length greater than 2 which begin with c_ic_j.

2. One or more polygrams of length greater than 2, and beginning with c_ic_j have been provided with individual Huffman codes; c_ic_j also has been provided with a Huffman code.

3. c_ic_j has not been provided with an individual Huffman code, and no polygrams of length longer than 2 characters have been provided with individual Huffman codes.

The following actions are desired when each of the above conditions applies:

1. The Huffman code for c_ic_j is entered into the encoded record, and the character following c_j is looked up in the first-letter table.

2. The character c_k following c_ic_j must be tested to determine whether $c_ic_jc_k$ is a trigram for which a Huffman code has been provided.

3. The Huffman code for c_i must be entered into the encoded record, and c_j looked up in the first-letter table.

To deal with these three conditions, then, three kinds of entries are provided:

1. Termination entry. Identical to a first-letter-table termination entry, except that (because there are 3 types of entries) the indicator is 2 bits long. The Huffman code is that for c_ic_j.

2. Continuation entry. Identical to a first-letter-table termination entry, with the above exception, and with the further exception that the address supplied is that of a third-letter table.

3. Retrace entry. This will be structured exactly like a termination entry; but the Huffman code it contains will be the code for the single character c_i.

Code conversion. Again, let c_i and c_j be conservative characters in the unencoded record. Suppose that c_i has a continuation entry in the first-letter table. Then the action required is that the computer's code for c_j be converted to another code. If c_j is a character for which a position has been provided in each second-letter table, then it will be converted to a code whose numeric value, $\underline{x}$, satisfies the condition $0 \leq x_j \leq n-1$, where $\underline{n}$ is the number of positions in each second-letter table. If $\underline{y_i}$ is the base address of the second-letter table for character $\underline{i}$, then $x_i + y_j$ will be that address in the second-letter table for character i which corresponds to the second letter j.

6.3.3 Encoding Example

The discussion up to this point will perhaps be clarified if an example is given. Suppose that we have source records written in an alphabet of 8 characters, $c_1, c_2, \ldots, c_8$. Suppose further that we have decided to provide Huffman codes for these 8 characters, plus the 14 digrams composed as shown in Figure 56.

Fig. 56: LIST OF DIGRAMS FOR THE ENCODING EXAMPLE

digram	components
d_1	c_2c_3*
d_2	c_2c_4*
d_3	c_2c_5
d_4	c_2c_7
d_5	c_3c_3
d_6	c_3c_4
d_7	c_3c_5*
d_8	c_4c_3
d_9	c_4c_4*
d_{10}	c_4c_5
d_{11}	c_4c_7*
d_{12}	c_6c_4
d_{13}	c_6c_5*
d_{14}	c_6c_7*

Suppose further that Huffman codes have been provided for certain polygrams of length greater than 2 which begin with the components marked with an asterisk in Figure 56.

Each individual character is represented in a 5-bit computer code, with the characters having the following values in that code:

character	value
c_1	4
c_2	5
c_3	6
c_4	8
c_5	10
c_6	14
c_7	18
c_8	20

Because c_2, c_3, c_4, and c_6 can begin polygrams for which Huffman codes have been provided, we will need four second-letter tables; and because only c_3, c_4, c_5, and c_7 are used as the second character of such polygrams, each second-letter table will contain 4 positions. Let us assume that our first-letter table begins at address 100; since it can end at position 120 (the maximum numeric value of a source-alphabet character is 20), the second-letter tables can begin at address 121. Our first and second letter-tables, then, will be as shown in Figure 57.

"T*" in Figure 57 indicates that a termination entry code would be found at that address; a* asterisk c means that at a*, a continuation entry table would be found. (Continuation entries in the first-letter table are followed by the address they contain); XXX means that the contents are irrelevant; and R§ stands for "Retrace code." Let us discuss how these tables would be used to encode a record. Let us assume we have the message $c_2c_3c_1c_4c_3c_5$ to encode.

Step 1. c_2 (= decimal 5) is used as a displacement in the first-letter table.

Step 2. At 105 in the first-letter table, we find a continuation entry and the address 121.

FIG. 57: EXAMPLE OF FIRST- AND SECOND- LETTER TABLES

Address	Contents	
100	XXX	First-letter table
...	...	
104	T*	
105	C: 121	
106	C: 125	
107	XXX	
108	C: 129	
109	XXX	
110	T*	
...	...	
114	C: 133	
...	...	
118	T*	
119	XXX	
120	T	Second-letter table corresponding to c_2
121	T*	
122	T*	
123	R§	
124	T*	
125	C:	2D-letter table corresponding to c_3
126	TC	
127	C:	
128	RC	
129	TC	2D-letter table corresponding to c_4
130	TC	
131	TC	
132	TC	
133	R	2D-letter table corresponding to c_6
134	T*	
135	R§	
136	R§	

Step 3. We now convert the code for c_3 (the second character of the message we are encoding) according to the conversion table shown in Figure 58.

FIG. 58: CONVERSION TABLE

Character	Old Value	New Value
c_1	4	4
c_2	5	5
c_3	6	0
c_4	8	1
c_5	10	2
c_6	14	6
c_7	18	3
c_8	20	7

Step 4. We now inspect the contents of the storage location whose address is: 121 (the number found in Step 2) + 0 (the new value of c_3).

Step 5. At location 121 we discover a termination entry. We therefore extract the Huffman code for c_2c_3 from this entry, and it becomes the first element of the encoded string.

Step 6. We now use the numeric value (4) of the next character (c_1) in the message as a displacement in the first-letter table.

Step 7. At location 104 we find a termination entry; we therefore extract the Huffman code for c_1 from the entry and place it in the encoded message, following the code just placed there.

Step 8. The numeric value (8) of the next character (c_4) of the message is used as a displacement in the first-letter table.

Step 9. At position 108 we find a continuation entry containing the address 129.

Step 10. The code for c_3 (the next character in our message) is converted (by the above table) to 0.

Step 11. At position 129 (= 129 + 0), we find a termination entry; we therefore place the Huffman code for c_4c_3 in the encoded message.

Step 12. The next (and final) character of the message (c_5) is used as a displacement in the first-letter table; a termination entry is found; the Huffman code is placed in the encoded message; and the encoding process is now complete.

It is now interesting to observe what would have occurred had we encountered, for example, the sequence c_2c_5. By the assumptions of our example, no Huffman code is provided for this digram; yet if this sequence had occurred either as the first two characters in the record, or immediately following a character which had yielded a termination entry, then the following would have happened:

1. c_2 would have caused the program to go to location 105 (in the first-letter table), where 121 would have been found.

2. c_5 would have been converted to 2, and at location 123 we find (Figure 57) a "retrace code." Here we would find the Huffman code for c_2. It would be inserted into the encoded record, and c_5 (its original code) would then be used as a displacement in the first-letter table.

6.4 Record Decoding

This section will discuss a method of decoding a string of bits, when the string is a series of variable-length Huffman codes. The decoding method described herein is based upon the use of what we may call the Decoding Dictionary (DD). Its maximum length (in bytes) will be the following:

$$\mathcal{L}_{DD} = 2^{r_1} b + \sum_{i=2}^{t} x_i 2^{r_i - r_1} + \sum_{i=1}^{q} i(n_i)$$

r_i = the set of Huffman code lengths

x_i = the number of Huffman codes of length r_i

t = the number of separate Huffman code lengths

q = the length (in characters) of the longest polygram for which a separate Huffman code is provided

n_i = the number of source symbols of length i

b = the number of bytes required to encode the number $\mathcal{L}_{DD}$ (the actual $\mathcal{L}_{DD}$, not the maximum as calculated above).

The above formula represents a maximum value, rather than an actual value, for reasons which will be evident after considering the decoding example given later. In most cases, the actual value of $\mathcal{L}_{DD}$ will be considerably less than this maximum.

6.4.1 Decoding Table Structure

The Decoding Dictionary, DD, may be conceived as having two parts. The first part, which we may call the Code String Segmentation (CSS) portion, in effect inspects the string of bits constituting the Huffman-encoded record, determines when a particular segment of the string constitutes a complete code, and supplies the address at which the source-alphabet equivalent of this code is to be found. The second part of DD may be called the Code Translation (CT) section. It supplies the source-alphabet equivalent of each Huffman code.

The method of constructing DD will be clear after the method of using it for decompression is described. Suppose that a group of source symbols consisting of single characters, digrams, and trigrams has been encoded in a Huffman code. The length of the shortest Huffman code used will be known. A number, n, or bits equal to this length is extracted from the left side of the coded stream. These bits are regarded as a number and added to the address of the first position of DD. At the address thus determined, a number will be found. If this number is less than or equal to the length (in bytes) of the CSS portion of DD, this means that the n leftmost bits of the Huffman-encoded stream do not constitute a complete Huffman code. Therefore, the next bit to the right in the stream is added to the number just found in CSS; this sum is added to the address of the first position of DD, and the number at this address (which will be somewhere in CSS) is inspected. As before, this number is tested to determine whether it is less than or equal to the length of CSS. If it is, the above process is repeated - the sum of the number just found, plus the next bit in the encoded stream, plus the address of the first position in DD, is formed; at this address a number is found; it is tested for "less than or equal to the length of CSS"; etc.

At some point (on the average, this will occur after the above process has been executed a number of times that is equal to the mean number of bits per source symbol, minus the length of the shortest code) a number will be found in CSS that is greater than the length of CSS. This will indicate that the end of the Huffman code has been reached; the number found in CSS is the address of the first byte of the source-alphabet equivalent of the Huffman code. However, since this equivalent may be a digram, trigram, etc., it will be necessary to know how many bytes are to be taken as the source equivalent. This is done as follows: Let n=the displacement of the first position in DD following CSS from the first position in DD. Then bytes $n, n+1, n+2, \ldots, n+q-1$ will each house a single character for which a Huffman code has been provided; the pairs of bytes beginning at $n+q, n+q+2, n+q+4, \ldots, m+q+x$, will each house a digram for which a Huffman code has been provided; the triplets of bytes beginning at $n+q+2$, $n+q+x+5$, etc., will each house a trigram. Let us assume that no source symbols longer than a trigram have been provided with a separate Huffman code; and let the last trigram be stored in the three bytes beginning at $n+q+x+y$. [$n+q+x+y+2$ then equals the length of DD.]

Now let r be the number (greater than the length of CSS) found in CSS. The following table indicates the length of the source symbol in bytes:

Condition	Length
$(n \le)\quad r < n+q$	1
$n+q \le \quad r < n+q+x$	2
$n+q+x \le \quad r < n+q+x+y$	3

(Again, the values $n+q+x$ and $n+q+x+y$ are defined at their maximum value. That these values will be considerably less than these maxima will be apparent below.)

6.4.2 Decoding Example

Let us assume a source alphabet of six symbols, composed of the four individual characters c_1, c_2, c_3, and c_4, and the two digrams c_1c_2 and c_2c_3. Let these six symbols have the probabilities shown in Fig. 59, and have assigned to them the Huffman codes shown there.

FIG. 59:
PROBABILITIES AND HUFFMAN CODES OF
CHARACTERS c_1, c_2, c_3, c_1c_2, c_2c_3, and c_4

source symbol	character equivalent	probability	Huffman code
s_1	c_1	.40	1
s_2	c_2	.30	00
s_3	c_3	.15	010
s_4	c_1c_2	.07	0111
s_5	c_2c_3	.05	01100
s_6	c_4	.03	01101

Let us assume that the DD array begins in byte 100. Its contents are the following:

byte no.	100	101	102	103	104	105	106	107	108
contents	02	10	11	04	12	06	08	14	15
byte no.	109	110	111	112	113	114	115	116	
contents	13	c_1	c_2	c_3	c_4	c_1	c_2	c_3	

(This example illustrates why the formula given above was for the maximus value of $\mathcal{L}_{DD}$. Note that the two digrams c_1c_2 and c_2c_3 require only three bytes (114-116) of storage, since the method described herein prescribes a first byte and a length. Hence, for example, the trigrams "the" and "her" could be stored in four bytes; these two and "ern" in five; these three and "new" in seven; etc.)

Now suppose that the message 000111101101 (the Huffman code for $c_2c_1c_2c_1c_4$) is to be decoded. Let the 12 bits of this message be B_1 through B_{12}, and let D_1 through D_5 be the five bytes of the decoded message. The decoding procedure is as follows:

1. $B_1=0$; the sum $100+0$ is formed.

2. At byte 100, 02 is found $(02<10)$.

3. The sum $100+02+B_2=102$ is formed.

4. At byte 102, 11 is found (11>10).

5. Since 111<114, byte 111 (c_2) becomes D_1.

6. B_3=0; the sum 100+0 is formed.

7. At byte 100, 02 is found (02<10).

8. The sum 100+02+B_4=103 is formed.

9. At byte 103, 04 is found (04<10).

10. The sum 100+04+B_5=105 is formed.

11. At byte 105, 06 is found (06<10).

12. The sum 100+06+B_6=107 is formed.

13. At byte 107, 14 is found (14>10).

14. Since 114=114, the two byte 114 (c_1) and 115 (c_2) becomes D_2 and D_3.

15. B=1; the sum 100+1 is formed.

16. At byte 101, 10 is found (10=10).

17. Since 110<114, byte 110 (c_1) becomes D_4.

18. B=0, the sum 100+0 is formed.

19. At byte 100, 02 is found (02<10).

20. The sum 100+02+B_9=103 is formed.

21. At byte 103, 04 is found (04<10).

22. The sum 100+04+B_{10}=105 is formed.

23. At byte 105, 06 is found (06<10).

24. The sum 100+06+B_{11}=106 is formed.

25. At byte 106, 08 is found (08<10).

26. The sum 100+08+B_{12}=109 is formed.

27. At byte 109, 13 is found (13>10).

28. Since 113<114, byte 113 (c_4) becomes D_5.

The process terminates at this point; the assumption is that the record began with a length counter, and that the value of this counter was 12.

6.5 Frequency Analysis and Alphabet Selection

The key element in the compression system is the selection of the symbol list which will be used in the source alphabet, each element of which will be replaced by a unique Huffman code. If the probability of occurrence of the characters of the alphabet (a,b,...z,0,1...9, etc.) at a given point were independent of the occurrence of the characters at other points in the record, then the maximum compression could be obtained through the use of a symbol list containing all and only the single characters. However, since this condition is violated in normal text (and, we assume, in bibliographic records as well) the inclusion of multiple character strings in the symbol list can lead to increased compression. The reason for this is that the frequency of the character string is not the product of the individual frequencies as would be indicated by independence. Thus, if the frequency of the string is higher than predicted, the Huffman code will be shorter than the sum of the individual character codes; and if lower, it will be longer.

But, at the same time, each increase in the size of the alphabet will increase the encoding and decoding efforts as well. Therefore, it appears desirable to investigate initially small increases in the size of the source alphabet by including a few two-character strings in the symbol list in addition to the basic single character set. In this way it will not be necessary to maintain a source alphabet of all digrams regardless of whether they appear. Because it is no longer a prefix code, encoding is no longer unique. This in turn can lead to an investigation of optimal encoding strategies. For the present, however, we are restricting our attention to the linear scan, maximum length character string procedure described earlier.

Thus, the procedure for selecting the symbol list is as follows. Initially the symbol list is set to contain only single characters, and the frequency of this occurrence is computed. From these frequencies the Huffman codes for the encoding alphabet can be computed.

6.5.1 Extension to Digrams

Next, the digrams (two-character strings) to be included are selected. To do this the single characters are arranged with descending frequencies: $c_1,c_2,c_3,\ldots,c_{64}$ with c_1 being the most frequent character, c_2 being the second most and so on. The n^2 digrams which can be obtained from the pairwise permutations are then included in the symbol list. If the characters are arranged in a matrix form as shown below, then the digrams to be selected are in the upper left square. Some of the digrams will have zero frequency because the count is obtained from a finite sample.

Ordered List of Characters

	c_1	c_2	c_3	⋮	c_n	⋮	c_{64}
c_1	c_1c_1	c_1c_2	c_1c_3	⋮	c_1c_n	⋮	c_1c_{64}
c_2	c_2c_1	c_2c_2	c_2c_3	⋮	c_2c_n	⋮	c_2c_{64}
c_3	c_3c_1	c_3c_2	c_3c_3	⋮	c_3c_n	⋮	c_3c_{64}
⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯
c_n	c_nc_1	c_nc_2	c_nc_3	⋮	c_nc_n	⋮	c_nc_{64}
⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯
c_{64}	$c_{64}c_1$	$c_{64}c_2$	$c_{64}c_3$				$c_{64}c_{64}$

Before proceeding with the process of generating the Huffman code for this symbol list, it is necessary to adjust the frequencies of the first n-characters because some of the occurrences of these characters will be encoded as digrams.

Consider a digram c_ic_j with frequency f. If this digram is to be included in the symbol list, we have to reduce the frequency of c_i and c_j by an amount equal to the expected number of times characters c_i and c_j will be encoded as a digram c_ic_j. The string of characters before each occurrence of c_ic_j is arbitrary. Hence, in the process of encoding, we will assume that we may end up in position 1 or 2 with equal probability.*

$$\text{----------}\quad {}^1c_i\,{}^2c_j\quad \text{----------}$$

c_i and c_j will be encoded as a digram only if previous encoding ends in position 1, which has a probability of 1/2. Therefore, the expected number of times c_i and c_j will be encoded as a digram is f/2. Hence, the frequency of c_i and c_j is reduced by f/2.

Varying n from 0 to 15, different symbol lists were obtained. Huffman code was generated for these lists and the average number of bits per character was calculated. The following shows the results of these codes.

n	Av. No. of Bits	Reduction	Remarks
0	4.796	20.6%	Only characters
5	4.446	25.9%	
10	4.364	27.3%	
15	4.343	27.6%	

---

*This assumption is inappropriate, especially where small numbers of multi-character strings are included in the symbol list.

The results clearly show that increasing n increases the reduction, but this increase is very small after n=10. On the other hand, as n increases, the number of symbols in the symbol list increases approximately in a square proportion. The computer time required for generating the code also increases more than linearly with the size of the symbol list. Hence, if only characters and digrams are to be included in the symbol list, it is not advisable to increase n beyond 15 or 20.

6.5.2 Extension to k-Grams

The following method of obtaining the symbol list and generating the code may be used to extend this idea to k-grams. It appears that the digrams may follow the same frequency order as that of the characters, so far as raw frequencies are concerned. This leads us to assume that trigrams, 4-grams, etc. will also follow the same order. Hence, the symbol list can be made to consist of all characters, n^2 digrams, n^3 trigrams,..., n^k k-grams, if we want to restrict ourselves to k-grams, where all these digrams, trigrams, etc. consist of the first n characters in the ordered list of characters. n will be made to vary from 1 to a certain feasible number.

This introduces the problem of adjusting the frequencies of single characters, digrams,..., (k-1) grams, if k-grams are to be included in the symbol list. Consider a case when only digrams and trigrams are to be included. One point should be noted at the outset: all the digrams that go to make all the n^3 trigrams are the n^2 digrams to be included in the list.

We start off with single characters being included in the symbol list. Next the n^2 digrams are included, for a certain value of n. In doing so, the frequencies of the constituent characters are adjusted in a manner explained above. When trigrams have to be included in the symbol list, for example a trigram $c_ic_jc_k$, it is obvious that digrams c_ic_j and c_jc_k already will be present in the symbol list. Hence the occurrence of the characters c_i, c_j and c_k is already taken care of while including the digrams c_ic_j and c_jc_k. Now it is necessary to adjust the frequencies of the digrams c_ic_j and c_jc_k only. This can be done in the same way as was done for digrams and characters. The difference is that if f is the frequency of $c_ic_jc_k$, then f/3 is subtracted from the frequencies of c_ic_j and c_jc_k. The coefficient 1/3 was obtained with the same considerations as were used with digrams. (In the case of trigrams, the required probability is 1/3.)

In general, the inclusion of k-grams is to be achieved step by step. First characters are included, then digrams, then trigrams,..., (k-1) grams. If k-grams having a frequency f are to be included, it is done by reducing the frequency of the constituent (k-1) grams by fk/k. After completing these adjustments, Huffman code can be generated for the new symbol list.

6.6 IBM 360 Implementation

The design of the compression system was then encoded for operation on the IBM 360/40. It was planned that the decoding routine would be established initially as a batch routine for debugging purposes and then established as an on-line component of the CIMARON system. The remaining routines for analysis and encoding would remain as batch oriented routines available for file building purposes. As yet, debugging of these routines has not been completed.

6.6.1 Analysis Program

The analysis program consists of a Fortran main program, ASMFOR, which calls a series of assembly language routines (CSECTS) to analyze the input data and establish the Huffman codes. These codes are then used by two other assembly language routines, ENCODE and DECODE, to encode and decode the source data and to record the times required and the compression obtained.

The main program, AMSFOR, calls the following assembly language CSECTS: PACK; COUNT; ADJUST; HUFTRE; HUFCOD; and IODISK. The function performed by each of the CSECTS and the parameters passed to them from the Fortran program are given in Fig. 60. The input parameters to these routines (as specified in ASMFOR) are:

a. the number of digrams to be included in the source alphabet set = DINUM (see Fortran listing);

b. the number of trigrams to be included in the source alphabet set = TRINUM;

c. the number of sample records from which the Huffman code is to be derived. This number = S, is specified as the number of times the DO loop 4 in ASMFOR is to be executed.

The output of the six CSECTS are four tables (arrays) called TABLE 1, TABLES, TAB and TREE, which contain the Huffman code for all source alphabets selected, and the binary tree (TREE) that will be used during decoding. These arrays have been stored on the disk ILR04 and given the DSNAMES ASHTABE1, ASHTABE2, ASHTABD1 and ASHTABD2, respectively.

The description of all the parameters is listed below:

TABLE1	- starting address of the table which stores all information concerning 1-grams.
TABLES	- starting address of the table which stores all information concerning digrams and trigrams.
TAB	- starting address of the array which contains all the 1-grams, digrams and trigrams included in the source alphabet.
TREE	- starting address of the binary tree that is used for code generation and decoding.
MARC	- starting address of a MARC record read in from tape.
TOTAL	- number of bytes in MARC record read into core.

MARKER - used as a flag when COUNT is called. Its value is zero when COUNT is called the first time (first sample record), and subsequently it is non-zero.

TAB2NM - denotes the number of half words used up by "table 2," viz., the first portion of TABLES which contain information on all digrams. The latter part of TABLES has trigram information in it. The division of TABLES into "table 2" and "table 3" takes place when COUNT is called the first time.

NUMBER - number of "table 3" entries, which is initially set to 6000, but increases until TABLES is filled up. NUMBER is increased every time that COUNT is called, since new trigrams are uncovered in the sample records.

DINUM - number of digrams to be included in the source alphabet set.

TRINUM - number of trigrams to be included in the source alphabet set.

TOTNUM - total number of alphabets (1-grams, digrams and trigrams) in the source alphabet set.

TRETOP - index of the cell within TREE which constitutes the top of the binary tree.

6.6.1.1 COUNT (TABLE1, TABLES, NUMBER, NARC, MARKER, TAB2NM, TOTAL)

COUNT is called by the Fortran program, ASMFOR, for each MARC record that is to be sampled for the frequency of occurrence of 1-grams, digrams and trigrams. The array NARC contains the record; TOTAL gives the length of the record (in bytes). TOTAL is returned by the subroutine PACK; ASMFOR then calls COUNT.

COUNT then samples the first 1000 bytes* of this record and divides the array TABLES into "table 2" and "table 3." TABLE1 stores the 1-gram count, TABLE2 the digram count, and TABLE3 the trigram count. The layout of the tables is shown in Fig. 61. Note that "TABLE2" and "TABLE3" are contiguous in core and constitute TABLES.

COUNT samples the first 1000 bytes of the first sample record for 1-grams and fill up TABLES accordingly. The COUNT field in TABLE1 stores the count for the particular 1-gram (alphabet), while the last byte of each entry stores the alphabet (using the internal code for alphabets). The POINTER field points to the head of a group of entries in TABLE2 where the digram counts for digrams starting with this alphabet would be stored. Thus, if the first sample (1000 bytes) containing a_1a_1 occurred n_1 times, a_1a_2 occurred n_2 times, and if no other digram starting with a_1 was encountered, then COUNT would take the following action. In the first scan of the sample text (scan for 1-grams only) a_1 may have been found to occur n_3 times. Then COUNT assumes that at most n_3 distinct digrams could have

---

*This limit was established to limit the total core requirement during the initial study. For regular operation it would be expanded to be consistent with the record size limitation of the rest of the CIMARON system. [ed.]

FIG. 60: DATA ANALYSIS CSECT FUNCTION AND PARAMETERS

DSNAME	CSECT	PARAMETERS	DESCRIPTION
ILR DEVOMOD (PHASE D)	PACK	NARC, TOTAL	Reads in a MARC record from tape, extracts its length from the first few bytes and passes the length and starting address as parameters to COUNT using NARC and TOTAL.
ILR DEVOMOD (PHASE 1)	COUNT	TABLE1, TABLES, NUMBERS, NARC, MARKER, TAB2NM, TOTAL	Counts the occurrence of all 1-grams, digrams and trigrams in each sample record that is given as an input to it. It is responsible for partitioning TABLES into "table 2" and "table 3." The number of distinct digrams and trigrams sampled is fixed by the size of the TABLES array.
ILR DEVOMOD (PHASE 2)	ADJUST	TABLE1, TABLES, NUMBER, TREE, TAB, TAB2NM, DINUM, TRINUM	Takes as input the tables that have been filled in by COUNT. It adjusts the counts for digrams to account for the fact that certain trigrams with that digram as prefix or postfix have been included as separate entities (source alphabets). Similarly, it adjusts the 1-gram counts to account for the digrams (containing the 1-gram) which have been included in the alphabet set. ADJUST is also responsible for selecting the DINUM most frequently occurring digrams and TRINUM most frequently occurring trigrams in the records sampled. All other digram and trigram entries in TABLES are cleared to zero. TAB is filled up with the 256 1-grams, DINUM digrams and TRINUM trigrams selected, along with their adjusted and weighed count, on the basis of which the binary tree will be generated.

FIG. 60: DATA ANALYSIS CSECT FUNCTION AND PARAMETERS (cont.)

DSNAME	CSECT	PARAMETERS	DESCRIPTION
ILR.DEVOMOD (PHASE 3)	HUFTRE	TAB, TREE, TRETOP, TOTNUM, TABLES	Takes as input the TAB array with the weighted counts (weighing factor = 1 for 1-grams, 2 for digrams, and 3 for trigrams), and constructs a binary tree (TREE). The method for constructing the tree is given in the detailed description of this CSECT. The index of the top cell of the tree is stored in TRETOP.
ILR.DEVOMOD (PHASE 4)	HUFCOD	TABLE1, TABLES, NUMBER, TAB, TREE, TAB2NM, TOTNUM	It takes as input the tree generated by HUFTRE, traces the path from each leaf of the tree to the top of the tree and thus derives the Huffman code for each source alphabet. Information concerning the number of bits in the code, and the code itself, are stored in TABLE1 and TABLES for all source alphabets.
ILR.DEVOMOD (PHASE Y)	IODISK	TABLE1, TABLES, TAB, TREE	Writes out the four arrays on disk (ILR04) as data sets with names ASHTABE1, ASHTABE2, ASHTABD1 and ASHTABD2. The first two will be used by the encoding routine, the latter two by the decoding routine. TRETOP is stored in the first half word of TREE, and TAB2NM in the first half word of TABLES. These two parameters are also required by ENCODE and DECODE.

FIG. 61: LAYOUT OF COUNT TABLES

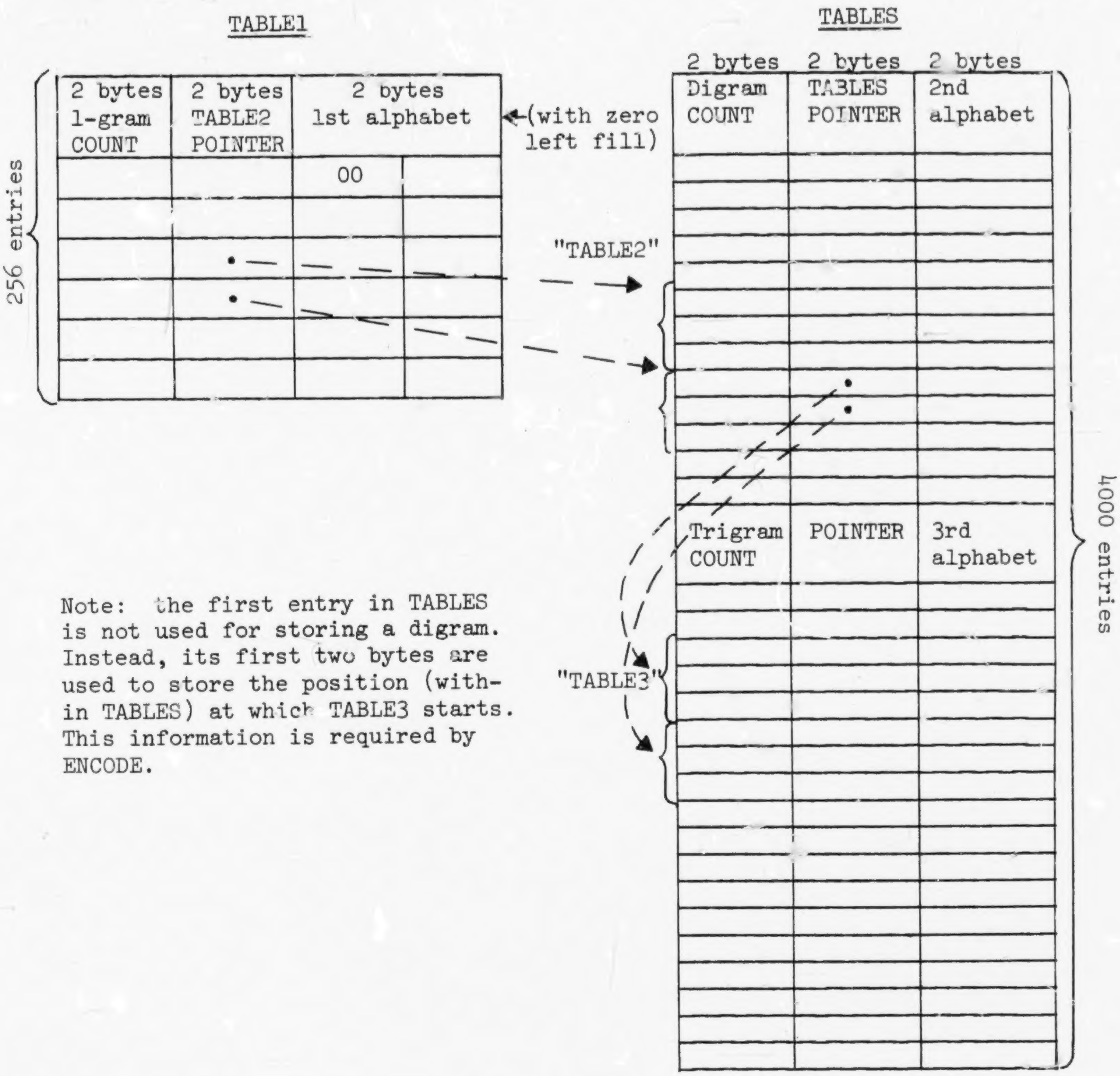


TABLE1
TABLES
2 bytes
1-gram
COUNT
2 bytes
TABLE2
POINTER
2 bytes
1st alphabet
(with zero
left fill)
00
256 entries
"TABLE2"
2 bytes
Digram
COUNT
2 bytes
TABLES
POINTER
2 bytes
2nd
alphabet
Trigram
COUNT
POINTER
3rd
alphabet
"TABLE3"
4000 entries

Note: the first entry in TABLES is not used for storing a digram. Instead, its first two bytes are used to store the position (within TABLES) at which TABLE3 starts. This information is required by ENCODE.

occurred with a_1 as the first alphabet, since n_3 entries are reserved in TABLE2 for digrams starting with a_1. The TABLES count field would show n_3, byte 6 would have a_1, and the pointer field would point to the head of a group of n_3 words in TABLE2 (see Fig. 61).

In the second scan of the sample text, all the digrams are counted and their count stored in TABLE2. After the second scan, the TABLE2 entry pointed to by the a_1 entry would have n_1 in the "count" field, and b_1 in the "alphabet" field. The next entry would have n_2 and b_2 respectively in the above fields. Since the other entries in this TABLE2 group are empty, they may be used for storing new digrams encountered in other sample records.

After scan 2 (scan for digrams) 1000 entries in TABLE2 would have been reserved (by pointer in TABLE1 entries), though many of them would be empty since digrams would have repeated in the text. However, if a particular character did not occur in the sample of 1000 bytes, the TABLE1 "count" field for it would be zero and no TABLE2 entries would be reserved for digrams. What happens if this character occurs in another sample record? To solve this problem, we reserve five entries for each of the characters that did not occur in the first sample, and the "pointer" field in TABLE1 set appropriately.

Thus, if N of the 250 1-grams did not occur in the first sample, the length of TABLE2 would be (1000 + 5N) entries. This quantity is stored in TAB2NM, and TABLES assumed to start immediately after this as shown in the figure.

In the third scan (for trigrams) the procedure for setting up pointers in TABLE2 is similar to that used for TABLE1, and the number of entries in TABLE3 reserved for trigrams starting with a_1b_1 equals the "count" for a_1b_1 as shown by the "count" field in its TABLE2 entry. It is apparent that 1000 TABLE3 entries would have been reserved after scan 3 of COUNT. Since there may still be space in TABLES to store more trigrams, NUMBER is used to store the current size of TABLE3. COUNT then returns control to ASMFOR.

MARKER is set to a non-zero number so that in subsequent calls to COUNT this subroutine observes that the first sample has been taken and that TABLES 2 and 3 have been carried out of TABLES.

In any other call to COUNT after the first one, the text is scanned twice as before, with this difference: if any digram occurs in the text which was not encountered in earlier samples, there would be no TABLE entry for it. If a_1b_i had not occurred before, then none of the entries in the group of n_3 reserved in TABLE2 would contain b_i in the "alphabet" field. If, however, there is space in this group for more digrams, b_i is entered in the appropriate field, and the count is set to 1. It is necessary to reserve some space in TABLE3 for trigrams starting with a_1b_i. NUMBER indicates the current size of TABLE3. Hence, five TABLE3 entries in the available space in TABLES are reserved and "pointers" set to point to the start of this group. NUMBER is also increased by 5 (or 30 bytes) to show the increase in TABLE3 size. If all space in TABLES is used up, it follows that trigrams with any digram (as prefix) not encountered in previous samples cannot be counted, and these are discarded. Similarly, since there can be 256^2 digrams and 256^3 trigrams (distinct), it is possible that within any group of entries reserved we may run out of space. These digrams or trigrams are discarded. With the size of TABLES equal to 24000 bytes, a maximum of 4000 distinct digrams and trigrams can be sampled and their counts stored.

6.6.1.2 ADJUST

This program is used to adjust the counts entered in TABLE1 and TABLE2 for 1-grams and 2-grams. Thus if a_1 occurs p_1 times in the text, and if, of the n_2 digrams selected, the total number of times that a_1 occurs in them is q_1, then the count for a_1 must be changed to (p_1-q_1). Similarly, the digram count must be adjusted to account for the count of those trigrams which include the digram as part of its body (prefix or postfix).

The first step is to select the n_3 trigrams which occur most frequently in the text. As mentioned earlier, TABLE3 contains the trigram count and is divided into groups whose pointer is stored in TABLE2 with the digram that is a common prefix of all the trigrams in this group.

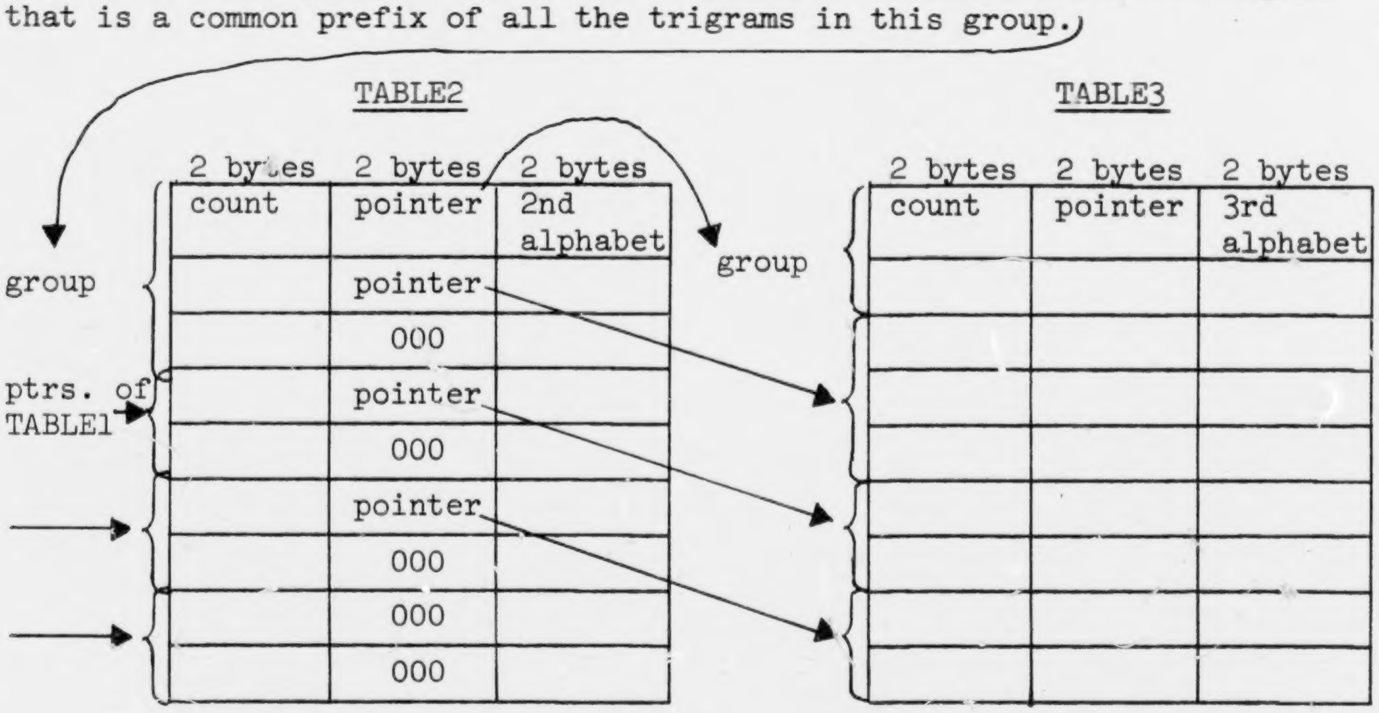
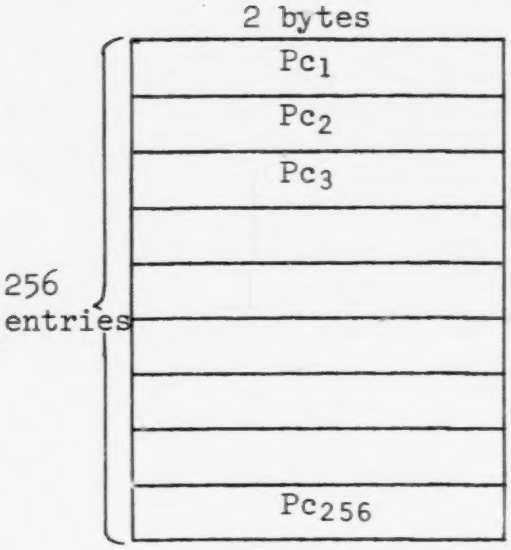


TABLE2
2 bytes
2 bytes
2 bytes
count
pointer
2nd alphabet
group
pointer
000
ptrs. of TABLE1
pointer
000
pointer
000
000
000
TABLE3
2 bytes
2 bytes
2 bytes
count
pointer
3rd alphabet
group

The program sorts out TABLE3 in a particular manner by sorting each group within TABLE3 in descending order of "counts" entries in TABLE3. Then it is necessary to select the highest n_3 trigrams. This is done as follows:



COMPAR

Each group in TABLE3 may be looked upon as a stack, the highest item being on top of each stack. Thus, if all the tops of all the stacks are compared, the highest member can be found. Now that stack in which the selected trigram occurred is popped up so that its second member comes on top.

Once again the top members of all stacks are compared to select the second most frequently occurring trigram, etc., until n_3 items have been selected. The 2-byte array COMPAR is used to store the pointers to the current tops of all the stacks, viz., the items that are to be compared to select the next trigram. Note that the pointers in TABLE2 still point at the highest items of each group in TABLE3. Thus, starting with the COMPAR pointers identical to the pointers in TABLE2, the COMPAR pointers for the selected stack (whose member was the highest) are incremented by one. When n_3 items have been selected, all non-selected entries in TABLE3 are cleared to zero. This is done in the following manner. The items in group 1 in TABLE3 between the pointer in TABLE2 and Pc_1 in the COMPAR array are to be retained and those between Pc_1 and Pc_2 (viz., the pointer to the start of the second group) are to be "zeroed." Similarly, those between Pc_2 and P_3 (including Pc_2), Pc_2 and Pc_3, etc. are all cleared to zero.

The next step is to find out those groups in TABLE3 which have all zero entries. The corresponding pointer for that group stored in TABLE2 is also cleared to zero.

Next the counts for some of the digrams in TABLE2 have to be adjusted to account for the inclusion of the trigrams which have that digram as a prefix or postfix, e.g., if a_1a_5 is counted d_1 times in the text, then the Pc_2 in TABLE2 points to the group of trigrams starting with a_1a_5, so the counts for all the non-zero trigrams in that group are added, divided by 2,* and subtracted from the count for the digram. This is repeated for all entries in TABLE2. Adjustment is necessary only for those digrams whose pointers are non-zero. Next, it is necessary to look for trigrams of the form $a_j a_1 a_5$ since the count for these must also be subtracted from the count for $a_1 a_5$. This task is less methodical to execute since the TABLES were ordered according to prefixes rather than postfixes. Postfix adjustment is done in the following manner: TABLE2 is searched for all entries having its "alphabet" portion equal to a_1. These are the entries for all digrams of the form $a_j a_1$. Next, the "pointers" in all these entries are used to locate the group of trigrams of the form $a_j a_1 a_k$. These entries are searched for entries having "alphabet" equal to a_5. All these "counts" in TABLE3 are added, divided by 2, and subtracted from the count for $a_1 a_5$. This procedure is repeated for all TABLE2 entries.

---

*If a_1a_5 occurred d_1 times, then for each occurrence of it a trigram $a_ia_1a_5$ and another $a_1a_5a_i$ must have been counted. Thus $\sum a_1a_5 = \frac{1}{2}[\sum a_ia_1a_5 + \sum a_1a_5a_i]$ and, hence, when adjusting for the digram count, the trigram count must be divided by 2.

Now the single character (TABLE1) entries are adjusted similarly to account for all 1-gram counts which are contained in the digram selected. N_2 digrams are selected from TABLE2 in an identical manner to the selection procedure for N_3 trigrams, the remaining entries being cleared to zero.

Now if the count for a_7 is p_7, and if a_7 occurs in digrams of the form $a_7 a_j$ or $a_k a_7$ in TABLE2 p_7 times, the adjusted count is $(p_7 - \frac{q^7}{2})$. The same procedure is followed. All digrams having a_7 as prefix are grouped together and their counts are added and subtracted from p_7. Next TABLE2 is searched for all entries with "alphabet" equal to a_7; their counts are added, divided by 2, and subtracted from p_7.

Next, all the counts in TABLE1, TABLE2 and TABLE3 are multiplied by their appropriate weighing factor, 1 for 1-grams, 2 for digrams and 3 for trigrams.

Next, it is necessary to identify all 1-, 2- and 3-grams to be included in the source alphabets and to sort their counts in increasing order to facilitate the setting up of a binary tree from which the Huffman code will be derived. This is done by setting up an array of TAB whose format is shown below.

TAB

256 + N_2 + N_3 entries

1 byte	3 bytes	2 bytes
IND	alphabets	Count
1	a_1 - -	p_1
2	a_1 a_2 -	q_1
3	a_1 a_2 a_3	r_1
1		
3		
2		

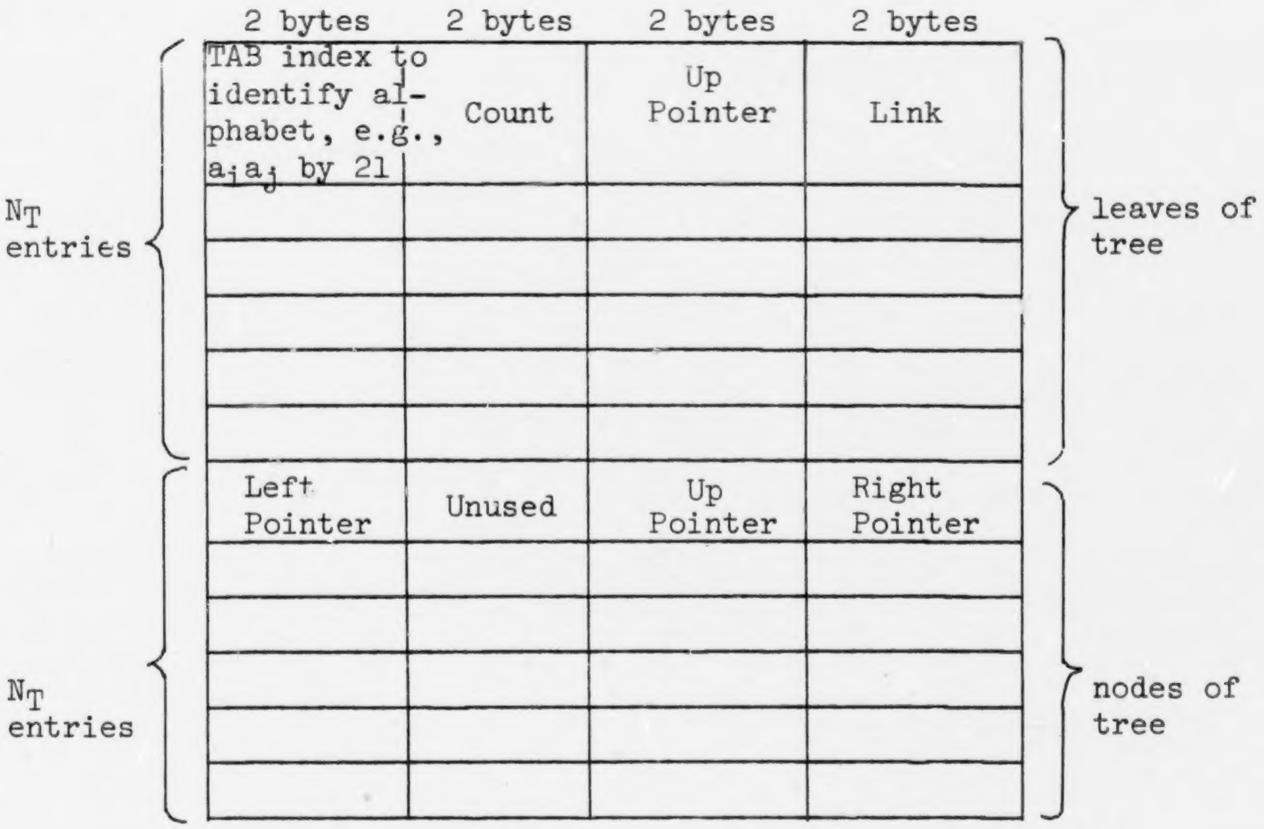
IND indicates number of alphabets in the source alphabet, viz., whether it is a 1-, 2- or 3-gram. The alphabet (k-gram) itself is stored in the next 3 bytes, and its count in the last 2 bytes. This array TAB is filled up using the non-zero entries in TABLES 1, 2 and 3. Then the TAB array is sorted in increasing order of counts. The TAB array will be used to construct the three and any source alphabet (1-, 2- or trigram) is identified in the tree, TREE, by its location in the sorted TAB Table which later will be used to enter the Huffman code into TABLES 1, 2, and 3; i.e., if the digram $a_i a_j$ occurs in the 21st location in the sorted TAB array when working with the binary tree, TREE, $a_i a_j$ will be known by the number 21, called the TAB index.

Note: to conserve space, the array TREE is used to store the pointers used during sorting, instead of having a separate array - COMPAR. TREE will be used by HUFTRE and, hence, it is cleared to zero after exiting from ADJUST back to ASMFOR.

6.6.1.3 HUFTRE

This program constructs the binary tree using an array TREE (see Fig. 62) from which the Huffman code for each source alphabet will be derived in the subsequent program phase.

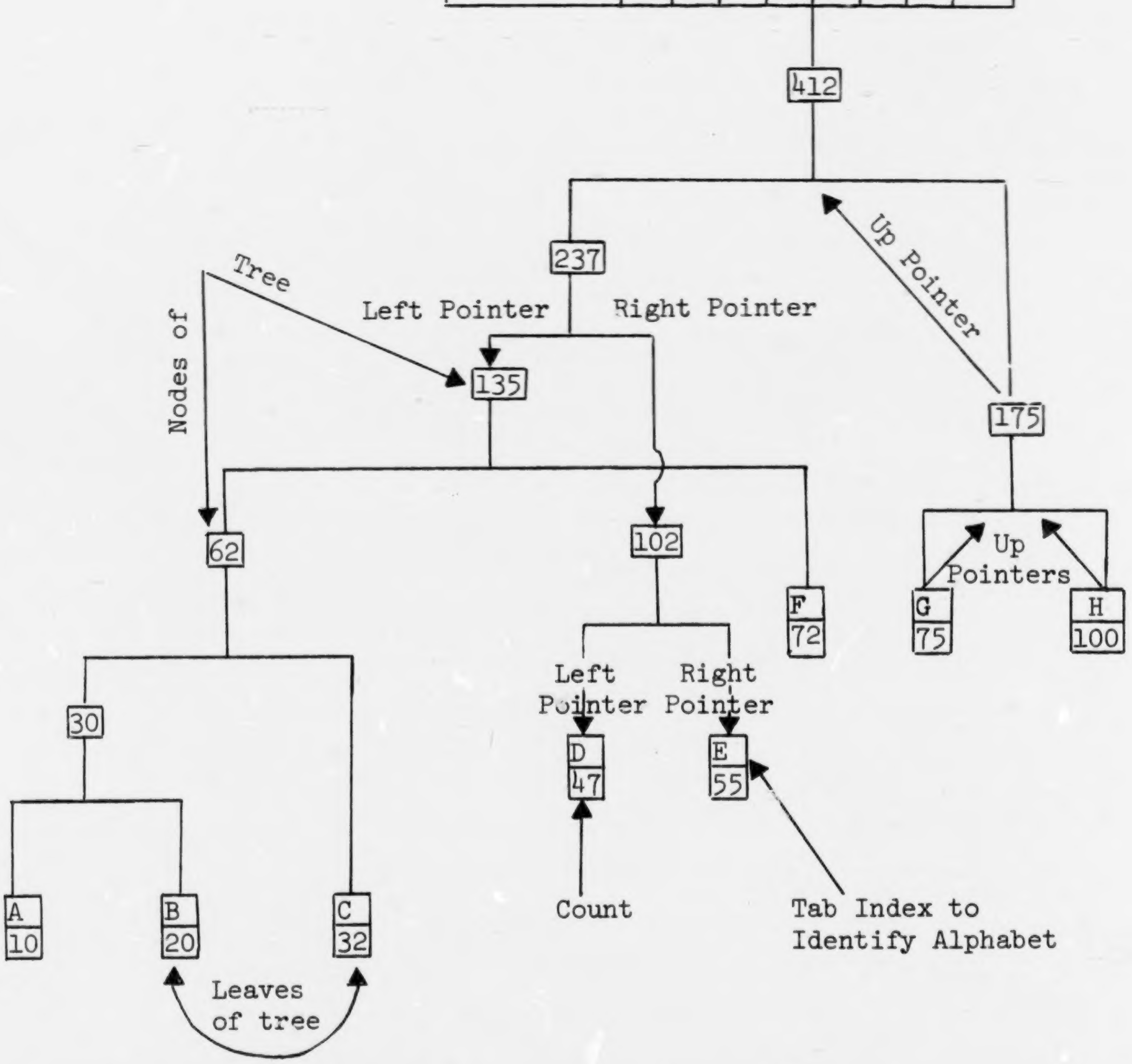
FIG. 62: STRUCTURE OF TREE ARRAY


2 bytes
2 bytes
2 bytes
2 bytes
TAB index to identify alphabet, e.g., $a_i a_j$ by 21
Count
Up Pointer
Link
N_T entries
leaves of tree
Left Pointer
Unused
Up Pointer
Right Pointer
N_T entries
nodes of tree

The contents of all sections of the TREE entries will be clear from a picture of the TREE that is set up, the linkages of the TREE being provided by the UP, LEFT, and RIGHT pointers. The purpose of LINK will be explained when the mechanism of the program is explained.

Assume that there are eight elements in the source alphabet (an element may be 1-, 2- or 3-gram) and their counts may be as follows:

Alphabets	A	B	C	D	E	F	G	H
Count	10	20	32	47	55	72	75	100


412
237
Up Pointer
Tree
Nodes of
Left Pointer
Right Pointer
135
175
62
102
F
72
Up
Pointers
G
75
H
100
30
Left
Pointer
Right
Pointer
D
47
E
55
A
10
B
20
C
32
Count
Tab Index to
Identify Alphabet
Leaves
of tree

The manner in which HUFTRE sets up this tree will now be described. Let N_T be the total number of source alphabets = $n_1 + n_2 + n_3$ (where n_1 is always 256). First, the counts for all N_T source alphabets and the appropriate TAB index to identify each source alphabet are copied into the top N_T entries of TREE using the TAB array set up in the last phase. All UP, LEFT and RIGHT pointers are "zeroed." All the N_T entries in the TREE are linked into a long list using the LINK field (see TREE format for top N_T words). Thus the first item on the list has the least count, and the bottom item the largest count. The LINK of the last item is cleared to zero.

The next N_T words of TREE will be used as nodes in the tree (see figure). Now we need an indicator to show how many cells in the lower N_T group are available for use when a new node is being formed. Hence a pointer AVSPAC (available space) is used to point to the first available cell in TREE, and when a node is formed, the AVSPAC is incremented by one. Another pointer called TOP is used to point to the top of the first N_T words which are linked as a list. TOP points at the current least count item. This item may be a single source alphabet or a set of two alphabets, etc. This is explained below.

The program proceeds as follows: TOP is used to obtain the least count alphabet (source) and using its link, the second lowest (i.e., second in the list) source alphabet. Their counts are added and stored in the second item. The top word is deleted from the list. Then TOP is made to point to the second item. Next, one cell is plucked off the lower N_T words, i.e., the top-most available cell, AVSPAC, is plucked off, and its left and right pointers are made to point at the top two cells as shown in the figure of the tree, the UP pointer in these two cells being directed toward the node cell (that was plucked off). AVSPAC is updated to point to the next available cell. Now the second cell which holds a count equal to the sum of the least two counts is treated as a single item. This count may not be the lowest in the list any longer, so the list is searched and a pair of links in these cells changed and the TOP made to point to a new cell which has the lowest count. Note that the location of the cells themselves in the TREE array are not changed, but only the LINKS adjusted.

Now the same procedure is repeated and a new node created with its LEFT and RIGHT pointers pointing to two nodes (or leaves) to its left and right (left denotes the cell with the lower count). In this way the entire tree is constructed. AVSPAC finally points to the topmost cell in the TREE. This tree can now be used to derive the Huffman code for all the source alphabets in the next phase, HUFCOD.

Note: The index of the top cell of the binary tree (header) is stored in the first half of the array TREE.

6.6.1.4 HUFCOD

This program phase, as the name suggests, is used to derive the Huffman code for the N_T source alphabets (1-, 2- and 3-grams) using the TREE constructed in the HUFTRE phase.

The TAB array is ordered in increasing order of count. Assume that the i^{th} alphabet in TAB (say a_i) is being encoded into Huffman code. Then the i^{th} entry in the tree contains information about this alphabet and is actually a leaf of the tree from which we must climb up the tree to the header, TRETOP. The item to the left is denoted by a 0, the one at the right by 1. The UP pointer of the leaf is used to get the node cell. The LEFT pointer of this is compared with the leaf address, and if they coincide, the leaf was to the left so the last digit for the code is 0; otherwise it is 1. Then, using the UP pointer of the node cell, we go one level higher up in the tree, and again the same test is carried on, the code is shifted in a pair of registers using the SRDL instruction, and each time the level is increased by one, a count is kept which denotes the number of bits in the

Huffman code for that source alphabet. When the header is reached, the register contains the H-code and another contains the levels in the tree traversed, i.e., the number of bits in the Huffman code.

When the Huffman code has been generated for a particular source alphabet, TAB is checked to see whether it is a 1-, 2- or 3-gram. If it is a 1-gram, say a_i, then the i^{th} entry in TABLE1 is fetched and its format changed to contain the Huffman code and number of bits of the code for a_i.

This new format for TABLES 1, 2 and 3 (from Fig. 61) is given below.

TABLE1

256 Entries

1 byte	1 byte	2 bytes	2 bytes
1st alphabet	Length of H-code	Huffman code	Pointer

TABLE2

1 byte	1 byte	2 bytes	2 bytes
2nd alphabet	Length of H-code	Huffman code	Pointer

TABLE3

1 byte	1 byte	2 bytes	2 bytes
3rd alphabet	Length of H-code	Huffman code	Pointer

If no 4-gram

There are 256 1-grams, N_2 digrams, and N_3 trigrams in the source alphabet. As each Huffman code is derived, the appropriate TAB entry is looked up to find out if it is a 1-gram, 2-gram or 3-gram. If it is a 1-gram, say a_i, then the Huffman code A number of bits in it are stored in the i^{th} entry in TABLE1 as shown above. If it is a digram, say $a_i a_j$, then the i^{th} entry in TABLE1 is inspected for its pointer, which gives the start of the group within TABLE2 holding the digrams starting with a_i. The entry, viz., H-code, and number of bits are entered at the place where the "alphabet" entry matches the digram's second alphabet a_j.

For trigrams, a search in TABLE2 for a match with the second alphabet is necessary. Using its pointer, TABLE3 is entered, and a search is made for a match with the third alphabet. The code is then inserted appropriately there.

6.6.1.5 IODISK (TABLE1, TABLES, TAB, TREE)

This subroutine is responsible for putting the tables created by HUFTRE and HUFCOD, viz., TABLE1, TABLES, TAB and TREE, on a disk for subsequent use by the ENCODE and DECODE routines. The sizes of these tables and their DSNAMEs are given below.

TABLE NAME	SIZE (bytes)	DSNAME	USED BY
TABLE1	1536	ASMTABE1	ENCODE
TABLES	24000	ASMTABE2	ENCODE
TAB	6000	ASMTABD1	DECODE
TREE	6000	ASMTABD2	DECODE

Note that the size of "TABLE2" within TABLES is stored in the first half word of TABLES, and the index of the header of the binary tree is stored in the first half word of TREE. These two pieces of information are needed by ENCODE and DECODE.

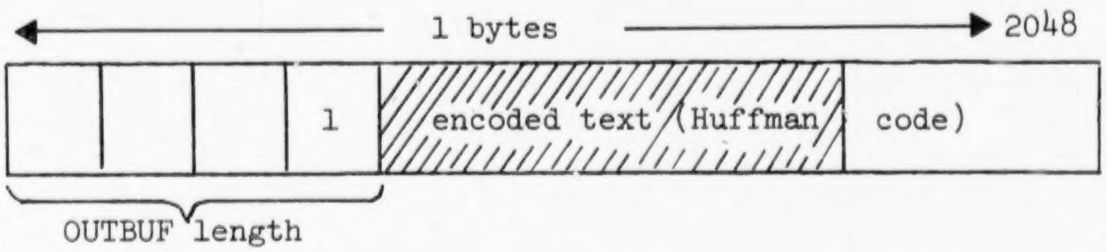
6.6.2 ENCODE

This program will be used as a batch processing job to take as input variable length alphanumeric records, encode them using the Huffman code derived for the chosen alphabet set (consisting of 1-grams, digrams and trigrams) and store the compressed records in an output file.

ENCODE makes use of TABLE1 and TABLES, which were put on disk as data sets with DSNAMEs, ASMTABE1, and ASMTABE2, respectively. Before processing the input tape (ALPREC), the routine does two GETMAINs and reads in the two tables from disk. An internal array, OUTBUF, of 2048 words is used to store the encoded record before transferring it to the output tape (HUFREC).

For each record from ALPREC, the following process is executed. The record is read into a core area reserved by a GETMAIN construction. The first word of the record gives the length of the record in bytes. The characters in the input stream are inspected from "left to right." Suppose the first three alphabets are $a_j a_k a_e$. First, the jth entry in TABLE is inspected. If the "pointer" filled is zero, it follows that no digram or trigram starting with a_j is included in the alphabet set. Hence, only a_j can be encoded. The Huffman code (and number of bits) are extracted from TABLE1 and inserted in a register. Encoding starts again at a_k and proceeds as before. Suppose the pointer in the jth TABLE1 entry is non-zero. We then enter TABLE2 (using the pointer) and search the group of entries to seek a match between a_k and the alphabet field. If no match exists, only a_j can be encoded and a_k is the new starting point for encoding. If the match succeeds, and the number of the Huffman code byte (i.e., byte #2) is zero, it follows that though $a_j a_k$ is not Huffman-coded, a trigram beginning with $a_j a_k$ is present. Using the pointer, we enter TABLE3 and seek a match with a_k. If it succeeds, $a_j a_k a_e$ can be encoded, and the code is present in TABLE3. Encoding begins beyond this trigram. If the match fails, only a_j can be encoded, and encoding starts next with a_k as the first input character. If byte 2 of TABLE2 was non-zero, $a_j a_k$ is a legal digram. This fact is noted, and next we seek to find a trigram beginning with it. If the pointer field is zero, or if a match in TABLE3 for a_e fails, then $a_j a_k$ is encoded from the code stored in TABLE2, and a_e becomes the start of the input stream. If the match with a_e succeeds, $a_j a_k a_e$ is encoded (using Huffman code stored in TABLE3). This process is repeated until the entire ALPREC text is encoded.

During the encoding process, the Huffman code is stored in a register and packed as each alphabet is encoded. When the register is filled, it is stored in OUTBUF. The first four bytes of OUTBUF are kept empty, and when the record has been encoded, the number of bytes in the total record (encoded record), including the first four bytes, is stored in binary format in the first four bytes as shown below.

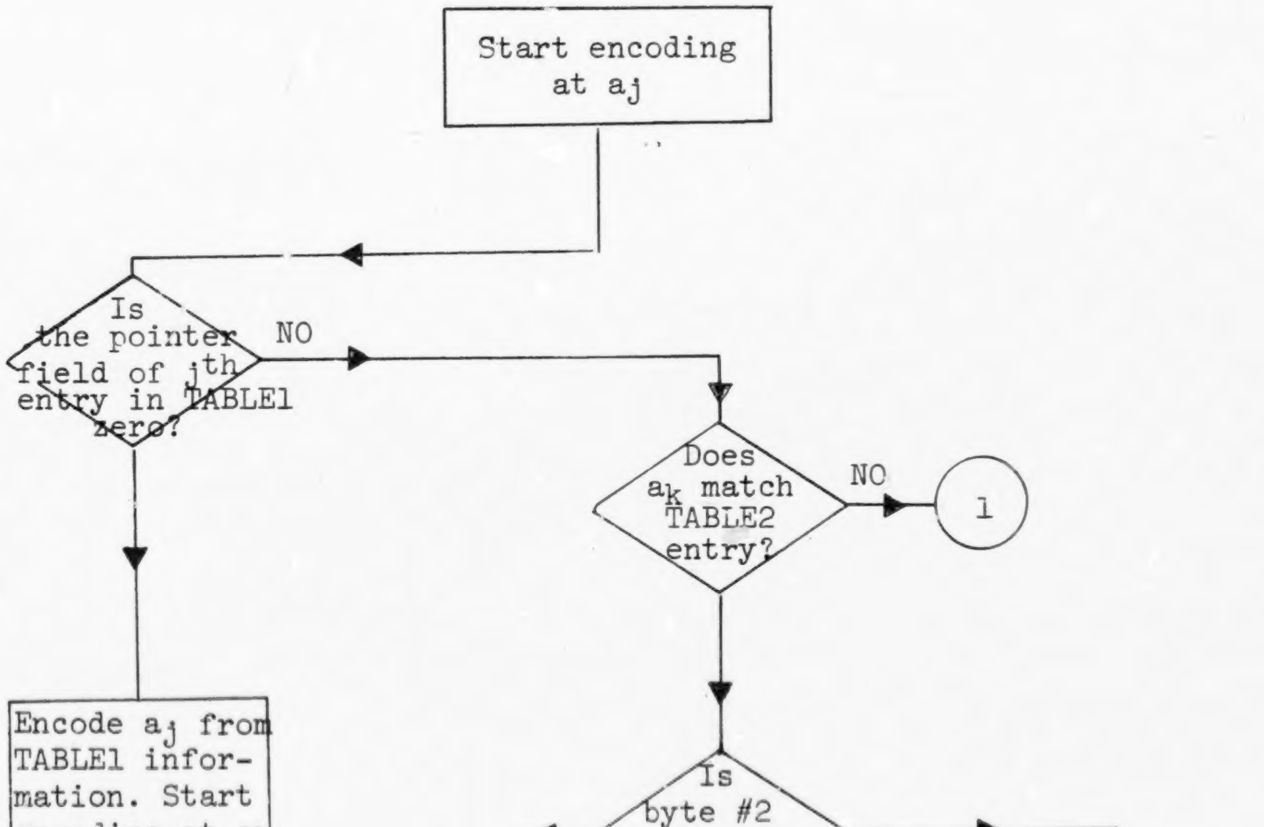


This array in then put on the output tape MUFREC. The next alphanaming record is read in and the same process repeated until every tape is encoded and stored in a compressed form on MUFREC. The encoding process is summarized in the flow chart below. The input stream is: $a_j a_k a_e a_m$ ... etc.

6.6.3 DECODE

This program will be called by a larger program and eventually will be called with two parameters. Register 0 will contain the length of the Huffman-coded record to be decoded, register 1, the starting address of the record in core. After DECODE is executed, it will return two paramameters via the same registers, register 0 containing the length of the decoded record (in bytes), register 1, the starting address of the area

FIG. 63: ENCODING PROCESS



Start encoding at aj
Is the pointer field of jth entry in TABLE1 zero?
NO
Does ak match TABLE2 entry?
NO
1
Encode aj from TABLE1 information. Start encoding at ak
1
2
Is byte #2 of TABLE2 entry zero?
Does ae match TABLE3 entry?
2
NO
Encode ajak from TABLE2 information. Start encoding at ae
YES
Is pointer field of TABLE2 entry zero?
NO
Encode ajakae from TABLE3 information. Start encoding at am
3
4
Does ae match TABLE3 entry?
NO
3
4

in core where the decoded record is stored.

However, this subroutine was tested as an independent unit. It read the tape, HUFREC, which contained the Huffman-coded record produced by ENCODE, and the encoded text for the first fifty records was printed out. The input records were loaded into a core area reserved by GETMAIN. The length of the record is stored in binary in the first four bytes and is used to detect the end of the decoding process. The decoded record is stored in another core area obtained by a GETMAIN.

DECODE requires two tables, TAB and TREE, stored on disk as ASMTABD1 and ASMTABD2, respectively. These are loaded into two areas also obtained by GETMAINs.

The process of decoding will be described now. Recall that the routine HUFTRE stored the index of the header of the binary tree in the first half word of TREE. The dividing of an input stream of bits begins from the header. When a 1 is encountered, we move to the right to the lower level node, and if a 0 is detected, we move to the left using the left pointer of the TREE entry. The TREE is traced until we reach a leaf of the TREE (detected by zero in last two bytes). The first two bytes of the leaf of the tree contains the index of a TAB entry. This entry is scanned. It may contain a 1-gram, digram or trigram. This k-gram is inserted into the area reserved for the decoded text. The decoding process is repeated using the bit of the input record following the point when the last decoding process concluded. The decoding starts at the top of the binary tree once again. The process is repeated until the input record is completed decoded. The decoded record is printed out as a file with DDNAME = PRINTOUT.

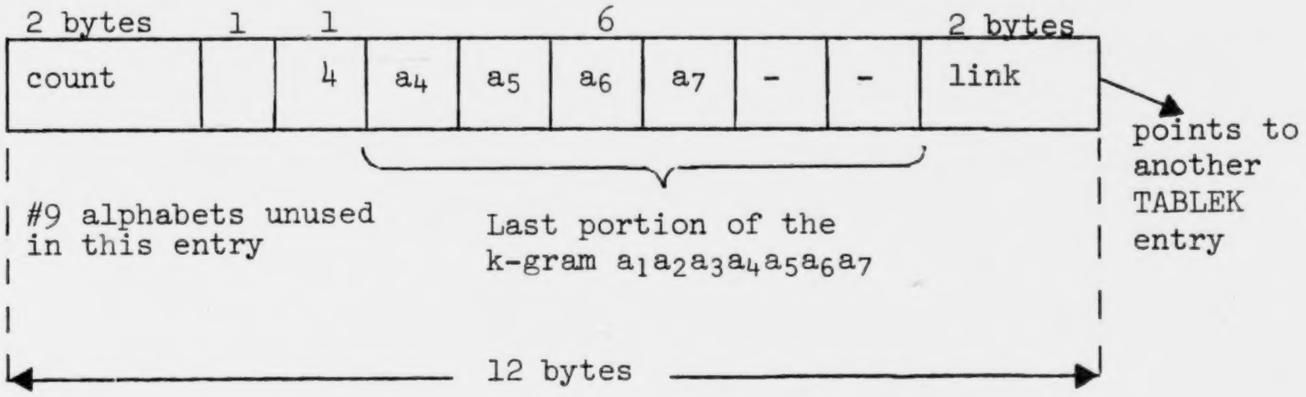
Space Condensation: The sizes of TAB and TREE are 6000 bytes each. However, the actual non-zero entries in these two tables would be a function of the number of source alphabets (i.e., the number of 1-grams, digrams and trigrams included). Thus, if N = number of source alphabets, the actual size of TAB would be 6N bytes, and the size of TREE = 8N (i.e., number of leaves of tree) + 8N (i.e., number of nodes in the tree) + 8 (top all used as pointer to the top of the tree).

Hence, if the value of N is also passed to DECODE, the GETMAINs above for TAB and TREE may be adjusted to accomodate only the non-zero portions of these tables. In this way a considerable saving in core area used by DECODE can be effected.

6.6.4 Extension of Program for Inclusion of k-grams (k>3)

It may be found necessary to extend the source alphabet set to include 4-grams, 5-grams, etc. A scheme is suggested for including up to 9-grams in the alphabet set. However, for any k-gram (k>3) the count, Huffman code, etc. are stored in a table, called TABLEK. It can contain anything from a 4-gram up to a 9-gram. Prior to the COUNT phase, the k-gram(s), which seem to occur often in the text, would have to know, or suppose we know, that "WILEY" occurs very frequently in a record set. We then may want to count its occurrence in a sample text and include it in the source set. Note that the ADJUST phase would not select from among these k-grams but include all of them in the source alphabet.

After COUNT had worked on the first sample record of 1000 bytes, TABLE1 and TABLE3 would have been initialized, and the latter split up into "table 2" and "table 3." Now the k-grams that we desire to be included could be read into an array, GRAMS, within the Fortran program, ASMFOR, via data cards. The k-grams would, of course, vary in size. Then an assembly language routine called KAGRAM would insert these k-grams into TABLEK, creating entries in TABLE1 and TABLES, if necessary, and setting the TABLES (actually "table 3") pointer to point into TABLEK. An AVSPAC (available space) indicator would be kept to indicate the first available entry in TABLEK while this insertion process is going on. The k-grams having the same first three alphabets would have entries in TABLEK linked together using the "link" field. The link of the last entry in any chain would be zero. The format of a TABLEK entry prior to ADJUST is shown below.


2 bytes
1
1
6
2 bytes
count
4
a_4
a_5
a_6
a_7
-
-
link
points to another TABLEK entry
#9 alphabets unused in this entry
Last portion of the k-gram $a_1a_2a_3a_4a_5a_6a_7$
12 bytes

The routine KAGRAM would insert the k-gram $a_1a_2a_3a_4a_5a_6a_7$ in the following manner. It would look at entry for a_1 in TABLE1 and, using the pointer, enter TABLE2. If a match for a_2 then failed, it would create a new entry and set up its pointer to a group of five entries available in TABLE3, as indicated by NUMBER, insert a_3 in the topmost entry, and set the TABLE3 pointer to point at AVSPAC in TABLEK. It would insert the last six characters as shown above. If the a_2 pointer field were non-zero, it would enter TABLE3 and search for a match with a_3. If present, it would examine its pointer field. If zero, it would create a TABLEK entry, and if non-zero, it would enter TABLEK and then use the link field to go down the chain. It would set the link of the last entry to point at AVSPAC, enter the k-gram in AVSPAC and update AVSPAC. If the match for a_3 failed in TABLE3, it would create an a_3 entry and a TABLEK entry as explained above.

After the k-grams were inserted, COUNT would be called S times, where S = number of sample records. The only difference in the execution of COUNT for any sample would be that if, while scanning for trigrams, the trigram's TABLE3 entry had a non-zero pointer, we would look for k-grams in the input stream and increment the "count" field of the TABLEK entry. The routine ADJUST remains unaltered, and, as far as it is concerned, TABLEK does not exist. k-grams are not sorted or "count adjusted." However, when the time comes to fill up TAB with the 1-grams, digrams and trigrams selected, ADJUST must go a step further and also look for non-zero TABLEK entries. These k-grams are also inserted into TAB. For any k-gram, two TAB entries are used and the format is shown below. TAB entry for k-gram $(k>3)$:

1 byte	5 bytes				
number of alphabets in k-gram	a_1	a_2	a_3	a_4	a_5
a_6	a_7	-	-	weighted count 2 bytes	

◄── 6 bytes ──►

The weighted count = count in TABLEK × k. When sorting TAB, care must be taken to recognize an entry for a k-gram so that both entries are treated as one. The TAB entries in the leaf of TREE for this k-gram points to the first byte of the first word of this TAB entry.

After MUFLOD has given abd the code, a TABLEK entry looks like the following:

◄── 12 bytes ──►

1	1	2	6	2
			alphabets	link
# of alphabets in k-gram	# of bits in Huffman code	Huffman code for k-gram		

ENCODE would remain unchanged except for the following alteration: if, while encoding an input stream, we enter TABLE3, we also seek a match with k-grams of the pointer if the TABLE3 entry is non-zero. We go down the chain of TABLEK entries pointed to by the TABLE3 entry, and if the match succeeds for any of the k-grams, that k-gram is encoded using the Huffman code in that TABLEK entry, and encoding starts again beyond the k^{th} alphabet in the input stream.

6.6.5 Program Operation and Results

The usefulness of Huffman coding can be measured best by noting the tradeoff between encoding and decoding time and the space compression achieved. The variables in this program are:

N_s = number of source alphabets, 1-grams, digrams and trigrams

N_d = number of digrams selected

N_t = number of trigrams selected

$N_s = N_d + N_t + 256$ (number of 1-grams = 256)

S = number of sample records on which the Huffman code for the N_s alphabets is derived.

C_{av} = average compression obtained per record

$$= \frac{(\sum_{i} \text{alphanumeric text length} - \sum_{i} \text{Huffman coded record length})}{\sum_{i} \text{alphanumeric text length}}$$

T_E = average encoding time per alphanumeric character (i.e., byte)

T_D = average decoding time per byte

L_{av} = average number of bits per source alphabet

$= \sum_{i=1}^{N_s} Li/N_s$, where L_i = number of bits in Huffman code for i^{th} source alphabet

L_{max} = maximum number of bits in Huffman code for a source alphabet = number of bits in least frequently occurring source alphabet

L_{min} = minimum number of bits in a source alphabet = number of bits in the most frequently occurring source alphabet.

Some of the functional relationships between these parameters are of interest as they focus attention on the space-time tradeoff in Huffman code compression techniques. These relationships could be obtained by varying certain parameters in the program, while keeping others fixed.

$L_{av} = f_1 (N_s, s)$

$C_{av} = f_2 (N_s, N_d, N_t, S)$

$T_E = f_3 (N_s, C_{av})$

$T_O = f_4 (N_s, C_{av})$

$L_{max} = f_5 (N_s, S)$

$L_{min} = f_6 (N_s, s)$

Among these parameters the input program parameters (which can be varied) are:

N_S = TOTNUM (see ASMFOR)

N_d = DINUM (see ASMFOR)

N_t = TRINUM (see ASMFOR)

S = Index of DO loop 4 in ASMFOR.

6.7 Results

6.7.1 Initial Method

Prototype programs for frequency analysis and for encoding were then written in the assembly language for the CDC 6400, COMPASS.* The method just discussed was used to obtain the source-symbol-alphabet and also the corresponding frequency of occurrence of each symbol. Huffman code was generated for the source-symbol-alphabet using the first method described before, and 200 records from the Library of Congress MARC II test tape were encoded in that code. The results of the actual encoding are shown in Fig. 64.

Two more methods were tried to select the source-symbol-alphabet and the corresponding frequency distribution. Both these methods are described below.

6.7.2 Second Method

As was explained, the probability of the previous encoding ending at position 1 or 2 of a digram $c_i c_j$ was assumed equal. Hence, if the digram $c_i c_j$ has frequency f, then the expected number of times the characters c_i and c_j will be encoded as a digram is f/2. Therefore, the frequencies of the constituting characters were reduced by f/2, but the frequency of the digram was left unchanged at f. But the same argument suggests that the digram frequencies should also be reduced to 1/2. Also by the same reasoning, the trigram frequencies should be reduced to 1/3 of their original frequencies. This additional feature was incorporated in this second method to obtain the frequency distribution for the source-symbol-alphabet. The result of this method is shown in Fig. 64. It clearly indicates that this method gives slightly greater reduction than the first method.

6.7.3 Third Method

This method is to skip all these arguments about probabilities and instead scan the records to get the actual count. The first drawback of this method is that each time a new source alphabet is tried, the records have to be scanned to obtain the actual frequency distribution. In addition to that, the reduction obtained by this method for any particular source alphabet was found to be less than that obtained by the first or the second method for the same source-symbol-alphabet.

Observing the results of these three methods given in Fig. 64, it is clear that the second method gives the maximum reduction. It also has another advantage in that the frequencies of all digrams and other polygrams can be obtained once for a given type of data, and these values can be used to select the optimal source-symbol-alphabet.

---

* The analytic error was found after these runs were made. However, the results are included as indicative of the potential of bibliographic compression, since correction of the error will improve the results. [ed.]

FIG. 64: COMPRESSION RESULTS*

Method No.	Avg. No. of Bits per Char.	Reduction
	For N = 10	
1	4.675	22.1%
2	4.658	22.4%
3	5.029	16.2%
	For N = 5	
1	4.800	20 %
2	4.732	21.1%
3	4.831	19.5%

6.7.4 Effect of Varying Source Alphabet Size

Although the second method is the most effective, a separate analysis had been run with the first method to determine the effect of varying the size of the source alphabet. Thirteen different source alphabets were created by varying the value of N from 0 to 60 in steps of 5. The results are given in Table 13. The following explains the significance of each column in Table 13.

1. N is the number of the highest frequency single characters used to obtain digrams (pairs) to be represented in the source alphabets.

2. NLAST gives the resultant number of symbols in the source alphabet. Since there are always 64 single character symbols in the source alphabet, NLAST - 64 gives the number of digrams actually used in the source alphabet.

3. The expected count for each symbol in one record.

4. The variance obtained from the count for each symbol.

5. Time taken in seconds to generate Huffman code, given the count for each symbol.

6. Minimum code length obtained in bits.

7. Maximum code length obtained in bits.

8. Average code length in bits per source character.

---

*Source-symbol-alphabet consists of characters, digrams and trigrams.

9. Compression obtained using that source alphabet.

10. Average time in milliseconds taken to encode one record.

11. Average time in milliseconds taken to decode an encoded record.

An interesting result that we did not anticipate is that the decoding time appears to decrease slightly with increases in the size of the source alphabet. Similarly, encoding times did not increase as anticipated. In future work it would be interesting to determine whether there is indeed a structural reason for this behavior. It should also be noted that the final two rows in the table have the same results. This is because increasing the value of N from 55 to 60 did not increase the number of digrams which are actually present in the 200 MARC II records which were processed. Finally, it should be noted that the frequency of occurrence of the symbols was computed using the same records which were subsequently encoded utilizing the codes obtained. Thus the compression obtained might be interpreted as a best possible compression ratio for a larger body of bibliographic records.

The two plots shown in Fig. 65 have been established using the data from Table 13. They show both the effect on the average number of bits required in the encoded representation for each character in the source string and the overall compression that is obtained utilizing this representation. These are shown as functions of the number of single letters whose pairs were considered for incorporation in the source alphabet. While much work remains to be performed, these results are most encouraging.

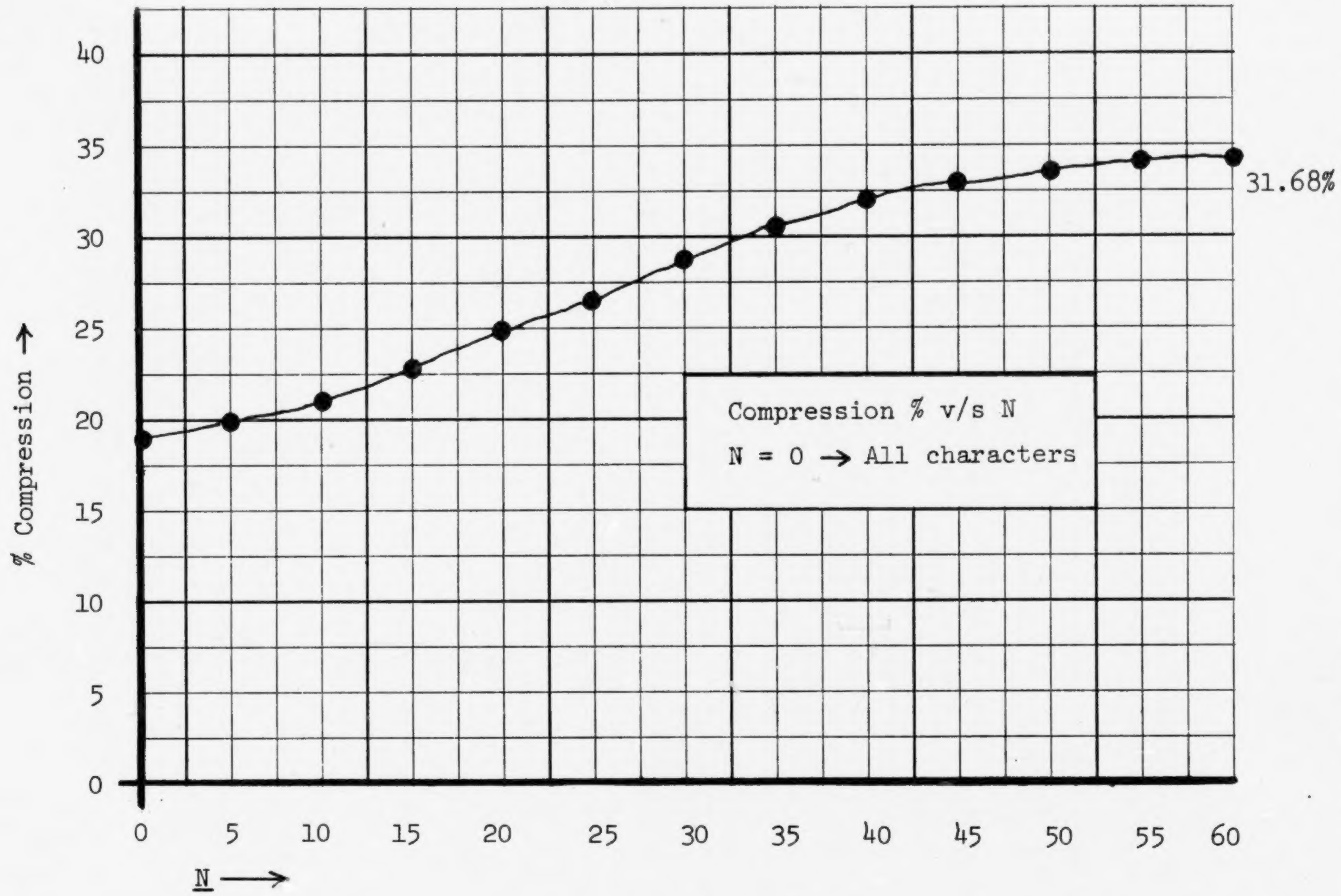
Table 13:

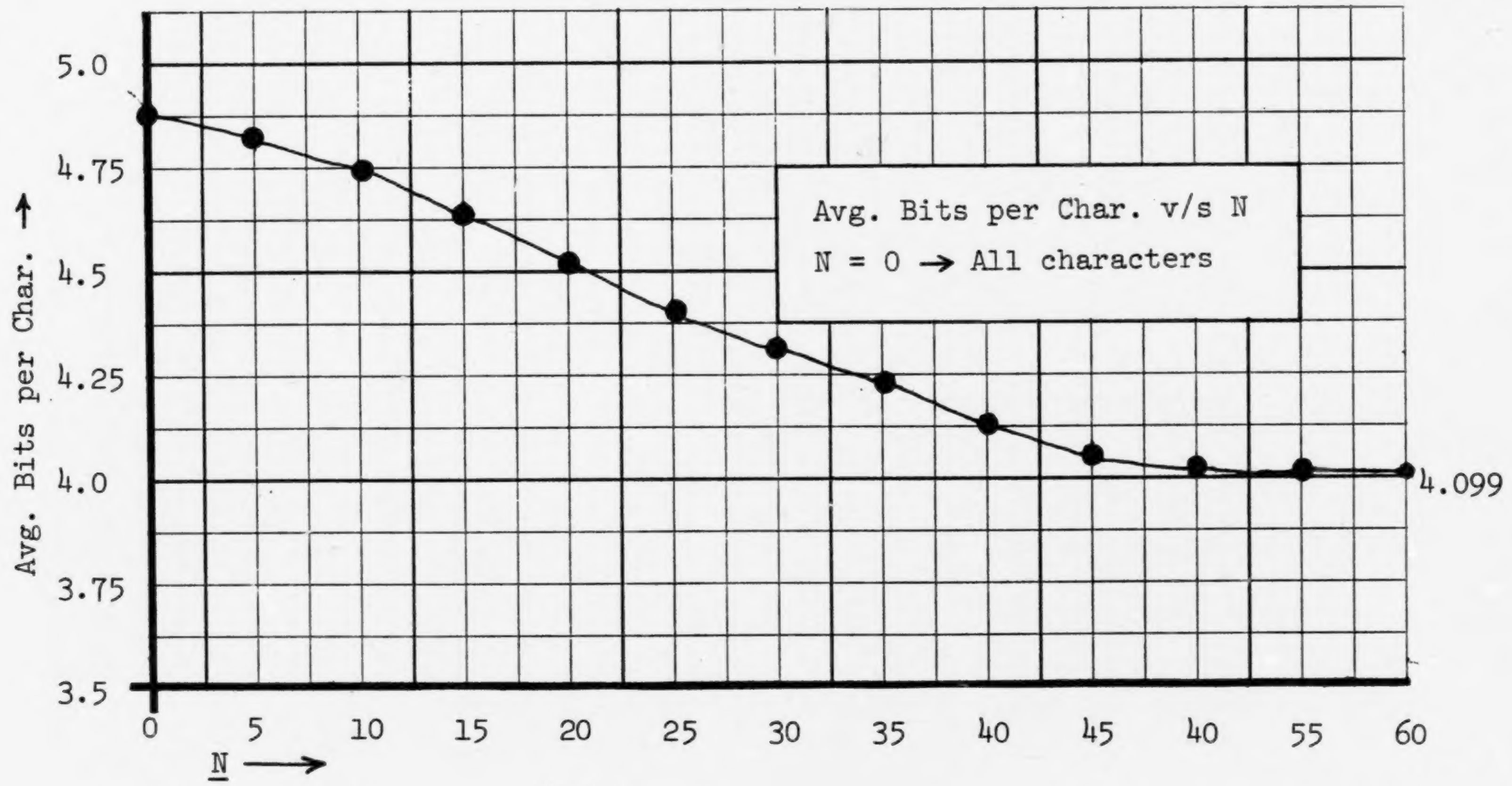
Results for Huffman-coding Procedure
for Characters and Digrams in 200 LC MARC II Records

1 N	2 NLAST	3 Exp.	4 Var.	5 Time to generate the code, sec.	6 Min.	7 Max.	8 Avg. bits/ char.	9 Compres- sion %	10 Avg. time to encode msec.	11 Avg. time to decode msec.
0	64	8.8906	210.4099	0.104	3	17	4.852	19.13	31	36
5	85	5.9979	55.8592	0.124	4	18	4.818	19.7	32	31
10	134	3.5511	29.1872	0.294	4	17	4.722	21.3	32	31
15	213	2.0669	14.7931	0.718	4	17	4.629	22.85	31	31
20	331	1.1944	6.4185	1.754	4	17	4.525	24.58	31	31
25	453	0.7823	3.1781	2.963	4	17	4.389	26.85	30	31
30	619	0.5270	1.8999	5.701	4	16	4.302	28.3	30	30
35	782	0.3886	1.3037	8.511	4	17	4.209	29.85	29	30
40	900	0.3777	1.1121	11.022	4	16	4.150	30.83	29	30
45	975	0.2987	1.0277	13.253	4	16	4.124	31.27	29	30
50	1045	0.2770	0.9637	14.744	4	16	4.106	31.575	28	30
55	1065	0.2715	0.9470	15.480	4	17	4.099	31.68	29	30
60	1065*	0.2715	0.9470	15.480	4	17	4.099	31.68	29	30

*No more digrams in the 200 records

FIG. 65: HUFFMAN CODING PROCEDURE RESULTS





7. SPECIFICATION FOR FORMAT TRANSLATION OF THE SANTA CRUZ FILE

By Jay L. Cunningham

7.1 Introduction

In the Spring of 1968, a joint task of the File Organization Project and the Operations Task Force Project* was begun. Its objective was to investigate the possibility of translation by computer program of the record format of the UC-Santa Cruz Library's machine catalog file into the prototype LC MARC II format then available. This file was originally created beginning in 1965 and conversion has continued since then. The size of the file is estimated to be in excess of 120,000 master records. The file is in a record format that was designed prior to the advent of the MARC I format established by the Library of Congress in 1966; however, the level of identification is similar.

Since then, the LC MARC project has grown in scope and momentum, and the MARC II format has been revised and accepted as the USA standard for bibliographic data in machine form. The MARC II format is intended as a "communications format" for the exchange of machine files among libraries and other agencies, and it has been adopted by the University of California as the exchange standard for library records. In addition, it is being used in augmented form, as the common processing format for the computer-based library systems of the University and for the University's Union Catalog Supplement. The latter format is called "UC MARC." As a result, the processing format of the File Organization Project was changed to the UC MARC format, and the objective of the automatic translation task became that of achieving this format. It was expected that the resultant file would be compatible with MARC II in structure, and the content would be a close approximation to exact MARC II tagging and other identifying conventions. Because of the volatile nature of the record standards during this effort, there are a number of intermediate programs and record formats which are used which would be dispensed with if a comparable effort were to be initiated at this time. However, the same functions would need to be performed if one is attempting an automatic translation of bibliographic records into a higher level of explicit content definition than is available in the source records. Therefore, the logic of this specific translation effort may be of general interest.

As an example of how "close" is "close": all personal name main entries and added entries will be identified and given the proper MARC II tag. However, the recognition algorithms at this time do not identify the sub-type of name. Each such name is therefore assigned the "single surname" indicator code, as a default. It is hoped that the deeper identification of this and numerous other imbedded data elements can take place in a later stage. Meanwhile, the file at the level of detailed identification attained in the routines specified herein should be adequate for book catalog production, search experimentation, and other uses not depending heavily on a "perfectly clean" data base at Level 1 MARC II. This file as described in this specification will have encoding at somewhere between Level 2 and Level 2 MARC.*

---

*This project is funded by the University of California.

The specification has three sections: instructions for translating the Santa Cruz original format to ILR Input Format, instructions for updating the resulting ILR Processing Format into the latest MARC II features, and lastly, some brief instructions on how to set up a UC-MARC record as an augmentation of the current MARC II.

When the translation study was initiated, the development of an input format and a program for translating this input into the ILR processing format had been completed. Therefore the translation objective was identified as that of producing translated records in the ILR Input Format in order to avoid duplicating the logic of the existing program. When the decision was made to change the processing record format, another sub-task was added to translate this format into the UC MARC format.

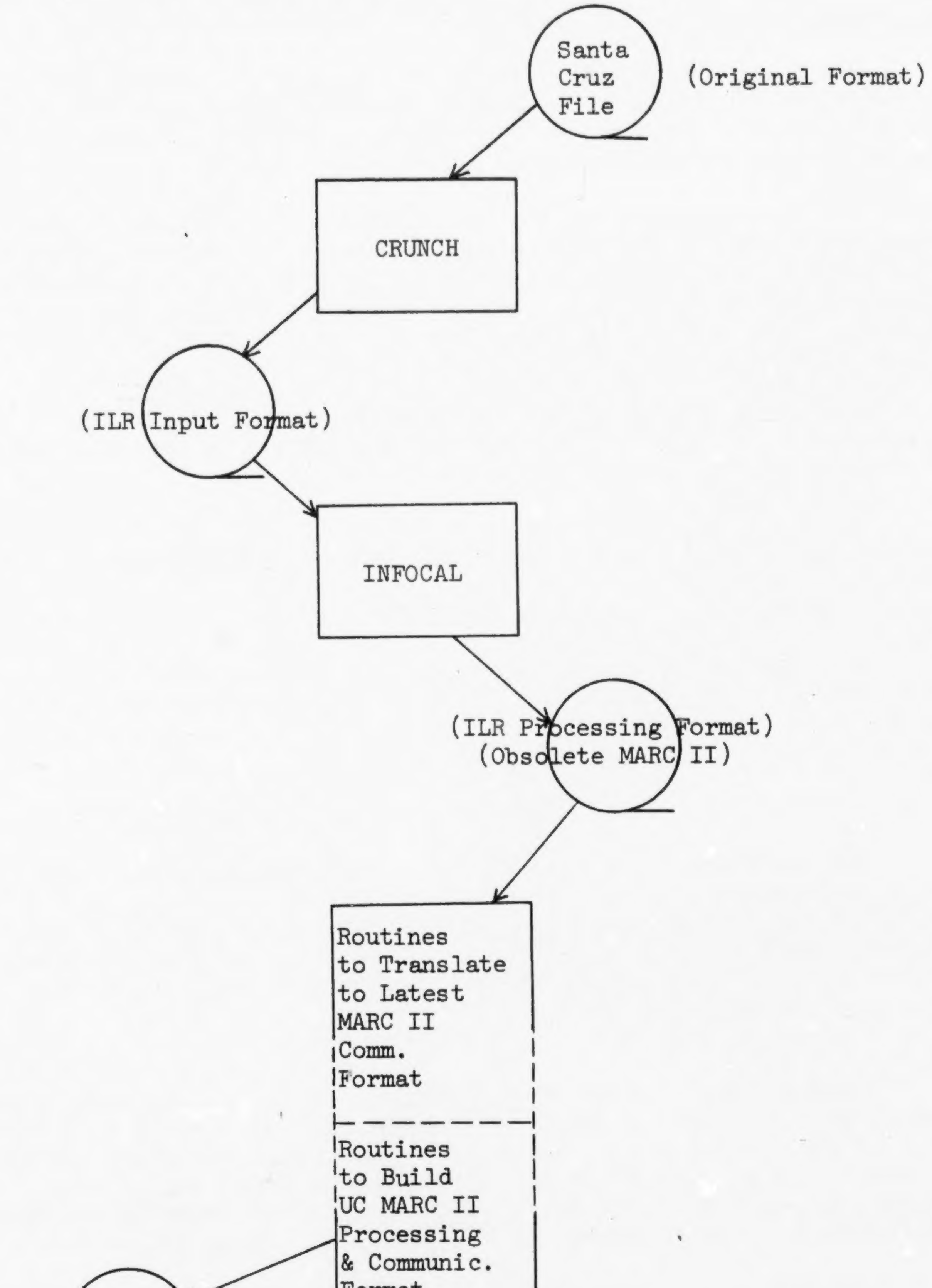
The following paragraphs cite the relation of the present specification to other documentation, for which no attempt to duplicate is made herein.

A diagram of the overall translation run is presented in Fig. 66.

---

*See: RECON Working Task Force, Conversion of Retrospective Catalog Records to Machine-readable Form: A Study of the Feasibility of a National Bibliographic Service, Washington, D.C.: Library of Congress, 1969, Appendix F.

FIG. 66:
SANTA CRUZ FILE TRANSLATION RUN


Santa Cruz File
(Original Format)
CRUNCH
(ILR Input Format)
INFOCAL
(ILR Processing Format)
(Obsolete MARC II)
Routines to Translate to Latest MARC II Comm. Format
Routines to Build UC MARC II Processing & Communic. Format (Common)
(UC MARC II Common)

7.1.1 Translation from Santa Cruz Original Format to ILR Input

The most complete specification of the Santa Cruz original record format is contained in the Key Punch Manual; University of California, Santa Cruz Library, July 1, 1965, rev. July 1, 1966.

The Santa Cruz original format is the source record format which constitutes the input to the CRUNCH program. The object record format is that which is produced as the output of the same program. This object record format is the ILR input record format.

The specification for the ILR input record format is contained in Appendix V, Part 2 of: Cunningham, Jay L., W.D. Schieber, and R.M. Shoffner. A Study of the Organization and Search of Bibliographic Holdings Records in On-Line Computer Systems: Phase I. Final Report. Berkeley, Institute of Library Research, University of California, March 1969.

The documentation of the equivalences produced by CRUNCH is contained in Section 7.2, below. For information on the meanings of the codes and the overall framework into which they fit, refer to the above-cited reports.

7.1.2 Translation from ILR Input to ILR Processing Format

From ILR Input Format the Santa Cruz records will be processed through the INFOCAL program written by ILR in May 1968. The output of this run will be records in ILR Processing Format, which is a modified MARC II format as of that date. (It is therefore obsolete in several respects.) See Appendix V-2 and Appendix IV of the above-cited Final Report of the File Organization Project.

7.1.3 Translation from ILR Processing Format to (Latest) MARC II

The specification of the MARC II Communications Format with which this translation program will comply is contained in: Subscriber's Guide to the MARC Distribution Service; Specifications for Magnetic Tapes Containing Monographic Catalog Records in the MARC II Format. Third Edition. Washington, Information Systems Office, Library of Congress, March 1969. 76 p.

Further modifications to the resulting file will be made by routines which adapt the MARC II Communications Format, as specified in Section 7.4, to the UC MARC Format, which is a common processing and communications format. The specification for UC MARC may be found in: Sherman, Don. University of California MARC Format. (Technical Paper No. 2) Berkeley, Institute of Library Research, June 15, 1969. 11 p.

Special instructions for building the UC MARC file from the updated MARC II INFOCAL output file are contained in Section 5 of the present paper.

7.2 Translation from Santa Cruz Original Format to ILR Input Format

7.2.1 Condition of Source File

Each logical record in the Santa Cruz original format is comprised of N tab cards. With the addition of the 8-character ILR-assigned record number ("Shoffner number"), the physical source (card image) record is now 88 characters in length. The object record--that which is to be input to INFOCAL--must be comprised of from one to fifteen 80-character images, in continuous string. The same record number (6 digits) plus card number (2 digits) is to be preserved from the original file.

Each record has an SC Accession Number which is to be preserved and placed in the special "*U" Shelf Key ("SK") field which is to be placed at the end of the record. It will be placed at the head of the field in character positions 1-6. (The SK field is fixed length--12 character prefix +26 characters. There is no $a delimiter at the beginning of the field; the number starts in character position one. This number is not used as the record number because it is not always unique over the whole file.

7.2.2 Format Translation Tables to Achieve ILR Input Format

Detailed equivalences for translation of Santa Cruz encoding to ILR input encoding follow in Figures 67-73. The emphasis in these figures is on the data elements and codes involved in the translation, rather than the program algorithms. For details on the program routines, see documentation for the TRANSCOF program in the following chapter.

7.2.3 Notes to Figure 67

1. This holdings code was defined prior to the advent of UC MARC. Since INFOCAL requires at least one JA-field to be input, and the routine has been written in the translate program, this field will appear as an 090 tag in the output record. The contents of the field should be purged in the program to revise Santa Cruz-INFOCAL, and replaced by a string corresponding to the revised UC MARC definition of a holdings field.

2. Due to an inadvertent programming error in INFOCAL, INFOCAL interprets the input code "RA" to mean "make title added entry" rather than "don't make title added entry," as originally intended. Therefore, for the purposes of the Santa Cruz conversion, "RA" will be supplied in the translated record when the symbol "L" is present in CT 200, Col. 69 of the source record. Do NOT insert "RA" if there is no CT 1XX card, however. The latter condition indicates a Title Main Entry, for which there is no such thing as a title added entry in the same form, by definition. A code "UB" will be inserted in clear cases of title main entry (absence of a CT 1XX card), to set the MARC indicator to "1", for "make title entry from Short Title." This will assure uniformity of encoding for title fields whether main or added.

FIG. 67:
FORMAT TRANSLATION TABLE I-FIELD SEGMENT

NAME OF FIELD OR SUB-FIELD	SOURCE FORMAT CODE OR CONDITION	OBJECT FORMAT CODE (IN ILR INPUT FORMAT)
(Tables are in logical sequence according to ILR Input data stream prescribed for INFOCAL)		
1. Master Record No. (MRN)	Col. 1-6 = ILR machine-assigned record no. ("Shoffner number"), same in each card in log. rec. Col. 7-8 = physical record no. (within logical record), in ascending sequence from Card 01.	Col. 1-6 = logical rec. no. (same as source rec.) Col. 7-8 = physical record no., in ascending sequence (same as source rec., for as many cards as needed.)
2. Publication Date 1, Publication Date 2, & Type of Date Key	CT (Card Type) 000, Col. 31-34 contains Year of Publication in Fixed Field format. (Card Type always appears in Col. 78-80 of the card image.) CT 300, variable position, contains variable date information, when condition is other than single date of publication. If record contains only one date, it appears only in CT 000.	(See Fig. 68) Build continuous string of codes beginning in character position 9 of card 01.
3. Bibliographic Level	CT 000, Col. 45: Other than 1 = Book 1 = Serial	DM (Monograph) DS (Serial)
4. Content Form	CT 500: Scan for key word	HB (Bibliographies)
5. Holdings(1)+	None (a dummy string is inserted)	JA/%%790% (Santa Cruz, stacks). TO BE REPLACED IN REVISION FOR UC MARC.

FIG. 67 (Cont.):
FORMAT TRANSLATION TABLE I-FIELD SEGMENT

NAME OF FIELD OR SUB-FIELD	SOURCE FORMAT CODE OR CONDITION	OBJECT FORMAT CODE (IN ILR INPUT FORMAT)
6. Main Entry in Body Indicator	Use same logic as for detecting presence of "C" Sub-field in Title (Rem. of Title Page Transcr.)	NA
7. Title Added Entry Indicator	CT 1XX is present; CT 200, Col. 69: Blank = make no title added ent. for title in this form L = make title added entry for this form of the title SEE ALSO TYPE OF ADDED ENTRY CODES (W-CODES) AND "*Q" CODE IN B-FIELDS, FOR HANDLING OF PARTIAL TITLE ADDED ENTRIES.	No action - no title traced same RA$^{(2)}$*
8. Language Code	CT 000, Col. 46-50: Blank = English (def.) For other codes, see p. 10 of Key Punch Manual, UCSC	SAccc% (For list of 3-character codes, see p. 183, Vol. 2 of The MARC Manuals. If blank, no action: INFOCAL default is "eng".
9. Translation Indicator	CT 1XX, Col. 69: T = Translator and other than Blank in Col. 46 Indicates likelihood that work is a translation. ALSO: must be two languages coded, at least, incl. "eng" as the default.	TA^{+} +NOTE: It is not known whether the language codes will be recorded in the order prescribed by MARC.
10. Type of Main Entry Code	If CT 1XX card present: Test first CT 1X0 000: CT 100 = Personal CT 110 = Corporate If NO CT 1XX card is present, this record has Title Main Entry:	UAP1 (Single Surname as default) UAC2 (Name, dir. order as default)

*See Section 7.2.3

FIG. 67 (Cont.):
FORMAT TRANSLATION TABLE I-FIELD SEGMENT

NAME OF FIELD OR SUB-FIELD	SOURCE FORMAT CODE OR CONDITION	OFJECT FORMAT CODE (IN ILR INPUT FORMAT)
	CT 200, Col. 69: (1st) "L" or Blank*	UATO (Title Main Entry; Set MARC Indic. to "1"
	"D"	UAUO (Uniform Title Main Entry Heading; Not Subject)
	NOTE: There may be both a CT D200 and a CT L200 or b̸200, in a single record. The "D" is usually first. If it is preceded by a CT 1XX, it is a supplied title (see Fig. 70). If it is NOT preceded by a CT 1XX, then assume Uniform Title Main Entry.	ALSO: insert "UB" code in string next; when "UATO" occurs. Do NOT insert "RA' code if there is no CT 1XX card present.

*According to the Santa Cruz Keypunch Manual, p. 17, there will be an "L" in Col. 69 if there are any added entries on a card having title main entry. There will be no "L" on such a card if it has no added entries. This means that in the case of cards having added entries, it may be impossible to identify the fact that the title main entry exists. In cases of ambiguity, the rule that the first CT 1XX card is the main entry will be followed. The condition is left ambiguous above since there is no assurance that Santa Cruz followed the keypunch conventions consistently over the whole file. The "L" is really superfluous for translation purposes. A "UB" code should be inserted in order to set the MARC indicator to "1", i.e., "make title entry."

11. Main Entry is Subject	Test first CT 1XX cp. to first 6XX CT for exact match on initial n characters, if date is 6XX.	UB
12. Type of Added Entry Codes:	For each CT 1XX card beyond the first 1XX card in the record:	String comprised of 2-character codes preceded by letter "W":
a. Other added entries	CT 100 = Personal: Col. 69 = J	PA (Single Surname; Alternative)

FIG. 67 (Cont.):
FORMAT TRANSLATION TABLE I-FIELD SEGMENT

NAME OF FIELD OF SUB-FIELD	SOURCE FORMAT CODE OR CONDITION	OBJECT FORMAT CODE (IN ILR INPUT FORMAT)
	=C	PA
	=E	PA
	=G	PA
	=I	PE
	=P	PA
	=T	PE
	=F	PE
	CT 110 = Corporate Col. 69: P = Publisher	CB (Name in direct order; Alternative) UO (Unif. Title Added Ent.)
	NOTE: NO W-CODE is required for Title-Traced-Same Added Entries ("L" in Col. 69). This function is taken care of by the "RA" code when the card is NOT title main entry.	
	N.B.: Problem on personal names coded as 110's.	
	For Title-Traced-Diff. Added Entries, W-CODE logic is: CT 200, Col. 69: P = Partial Title (added entry)	T1 (Secondary Title added entry) (default)
b. Series Added Entries	CT 200, Col. 69: S = Series Tracing and, when present: CT 500 containing a Series Note	(See Fig. 69.)
c. Subject Added Entries	CT 600-604: Scan for dates in string. If present: Personal Name Heading	P1 (Single Surname)
	Else:	00 (Topical) (default)
d. Subject Sub-divisions	CT 600-604: Scan for single hyphen	

FIG. 67 (Cont.):
FORMAT TRANSLATION TABLE I-FIELD SEGMENT

NAME OF FIELD OF SUB-FIELD	SOURCE FORMAT CODE OF CONDITION	OBJECT FORMAT CODE (IN ILR INPUT FORMAT)
	preceded by space	50 (General Subj. Subdiv.) (default)

SEE ALSO FIG. 70 FOR DISPOSITION OF VARIABLE FIELD INFORMATION ASSOCIATED WITH EACH OF THESE TYPES OF ADDED ENTRIES.

SPECIAL NOTE:

ORDER OF CODES IN W-CODES STRING: W followed by 1) all 2-character strings for Series Traced Same, 2) all 2-character strings for Subject Added Entries, 3) all 2-character strings for Other Added Entries, 4) all 2-character strings for Series Traced Differently. Within each sub-group, 2-character strings should be in same order corresponding to order in source record.

FIG. 68:
FIXED-LENGTH PUBLICATION DATES LOGIC

SOURCE CONDITION COMBINATIONS	DISPOSITION - ILR INPUR FORMAT		
	MOVE DATE TO COL. 9 - 12	MOVE DATE TO COL. 13 - 16	CODE FOR KEY*
CT 000 date present	X		BC
CT 300 date present - "C" for copr. (use first 4 digits after "C")		X	
CT 000 date present			BM
CT 300 date(s) present - multiple date span: (2 patterns) dddd-dddd	X (1st group of 4 digits)	X (2nd group of 4 digits)	
dddd-dd	X	X (same as above but expand to 4 digits)	
CT 000 date present			BM
CT 300 date(s) present - "open" entry: (2nd group of 4 digits missing) dddd-	X (omit hyphen)	No action - omit (INFOCAL will insert "9999")	
CT 000 date not present	No action	No action	BN
CT 300 date not present	No action	No action	
CT 000 date present- digits missing:	INSERT NORMALIZED SPAN		BQ
196-	X(1960)	X(1968)	
19--	X(1900)	X(1968)	
CT 300 not present			

*Insert in next sequential two character positions after last four digits of date element.

FIG. 68 (Cont.):
FIXED-LENGTH PUBLICATION DATES LOGIC

SOURCE CONDITION COMBINATIONS	DISPOSITION - ILR INPUT FORMAT		
	MOVE DATE TO COL. 9 - 12	MOVE DATE TO COL. 13 - 16	CODE FOR KEY
CT 000 date present - digits missing	INSERT NORMALIZED SPAN (as above)		BQ
CT 300 present (dt. of repro. or such)	(Disregard dates in CT 300 for Dates 1 & 2 purposes)		
CT 000 date present - (orig. date of pub.)		X	BR
CT 300 date present (dt. of reprint)	X		
CT 000 date present	X		BS
CT 300 date not present		Omit	

SEE ALSO PAGE 19 FOR DISPOSITION OF PUBLICATION DATES IN VARIABLE FIELDS.

FIG. 69:
LOGIC FOR TYPE OF ADDED ENTRY CODE - SERIES

SOURCE CONDITION	DISPOSITION - ILR INPUT FORMAT	
	ACTION	W-CODES; REMARKS
CT S200 is present; the first CT 500 present has string bounded by paren's. in initial portion of field. (NOTE: CT S200 is the series tracing if present; string in CT 500 is the series note.)	Compare length of string in CT S200: length of string inside ()'s in CT 500.	Set codes according to below:
	Lengths EQUAL: Scan for intervening period in CT S200. Period = evidence of Corporate Series Traced Same in CT S200. No Period = evidence of Title-Only Series Traced Same in CT S200.	C2 (Corp. in direct order; author not in main entry) (*A) NO W-CODE (*B)
	Lengths UNEQUAL: CT S200 = Corporate Series Tracing, Traced Differently CT 500 = Corporate Series Note, Traced Differently in form in CT S200.	C2 (Corp. in direct order) (*R) NO W-CODE (*D)
First CT 500 has string in (), but there is no CT S200 card.	None	NO W-CODE (*C) (Series note not traced)
NO CT S200 card present; NO CT 500 card with ()	None	NO series note in this record

SEE ALSO FIG. 72 FOR DISPOSITION OF VARIABLE FIELD INFORMATION ASSOCIATED WITH THESE TYPES OF ADDED ENTRIES.

FIG. 70:
FORMAT TRANSLATION TABLE A-FIELD SEGMENT

NAME OF FIELD OR SUB-FIELD	SOURCE FORMAT CODE OR CONDITION	OBJECT FORMAT CODE (IN ILR INPUT FORMAT)
1. Local Call Number	CT 000, Col. 1-30: Scan right to left fr. Col. 30 for 1st non-blank	Concatenate with end of W-CODES: Insert 1st "/" followed by call number
2. Main Entry Heading	CT1XX present (first 1XX card):	Insert 2nd "/" after end of call number, followed by author string.
	CT 1XX not present (i.e., title main entry):	Insert 2nd "/" followed by 3rd "/" in next sequential character position. Proceed to logic for CT 2XX.
Personal Name Sub-fields:		
a. Personal Name	CT 100, Col. 1-60: Scan left to right, test for numerals; test for 3 consecutive blanks, etc.	Insert "1F$_{16}$A" in next sequential character position following 2nd "/", followed by author surname.
b. Titles of honor	Scan for key word.	Insert "1F$_{16}$C" in character position preceding title of honor. Reposition element as needed.
c. Dates of birth, etc.	Scan for digits not immediately succeeded by non-blanks.	Insert "1F$_{16}$D" in character position preceding date element, following punctuation, if any.
d. Relator	CT 100, Col. 69:	Insert "1F$_{16}$E" followed by indicated abbreviation:
	J	j^{t}. author
	C	comp.
	E	ed.
	G	joint ed.
	I	illus.
	P	publ.

<table>
<tr><th colspan="3">FIG. 70 (Cont.):
FORMAT TRANSLATION TABLE A-FIELD SEGMENT</th></tr>
<tr><th>NAME OF FIELD OR SUB-FIELD</th><th>SOURCE FORMAT CODE OR CONDITION</th><th>OBJECT FORMAT CODE (IN ILR INPUT FORMAT)</th></tr>
<tr><td></td><td>T
F</td><td>tr.
ed./tr.
Concatenate after dates. This code and element should be preceded by a comma.</td></tr>
<tr><td>3. Sort Key Author Field</td><td>CT 1XX, Col. 69:
X = Sort Key Version of Author Name</td><td>Disregard. Field not carried to object record.</td></tr>
<tr><td>4. Uniform Title (Supplied)</td><td>CT 200, Col. 69:
D = Standard Title AND at least one CT 1XX</td><td>After end of 2nd "/" (for Main Entry), insert one "$" followed by the uniform title string.</td></tr>
<tr><td colspan="3">(THIS FIELD NORMALLY PRECEDES THE CT 2XX FOR TITLE STATEMENT, WHICH WILL BE CODED "I" OR "BLANK" IN COL. 69. THESE TITLE CODES ARE INDEPENDENT.)
(IF THERE ARE 2 OR MORE SUPPLIED TITLES--CODE "D"--ALL SUCCEEDING FIELDS WILL BE GIVEN W-CODE "U2" AND B-FIELD TAG "*Q". THEN MOVE STRING(S) TO B-FIELD REGION.)</td></tr>
<tr><td>5. Title Statement:</td><td>CT 200, Col. 69:
"L" or "BLANK"</td><td></td></tr>
<tr><td>a. Short Title sub-field</td><td>CT 200, Col. 1-60 (may be contin.)</td><td>Insert 3rd "/" at head of title string beginning in character position 1.</td></tr>
<tr><td>b. Remainder of Title</td><td>Algorithmic test for end of Short Title.</td><td>Insert 4th "/" between end of Short Title and beginning of Remainder of Title string. "/" goes after punctuation mark, if any, at end of Short Title (4th "/" must always be present.)</td></tr>
<tr><td>c. Remainder of Title Page Transcrip.</td><td>Algorithmic test for end of Title Page Transcrip.</td><td>Insert "1F$_{16}$C" followed by string. This delimiter is present only</td></tr>
</table>

FIG. 70 (Cont.):
FORMAT TRANSLATION TABLE A-FIELD SEGMENT

NAME OF FIELD OR SUB-FIELD	SOURCE FORMAT CODE OR CONDITION	OBJECT FORMAT CODE (IN ILR INPUT FORMAT)
		when data is present. Edition field in this position is not regarded as part of this sub-field. Coding is independent.
6. Series Tracing	CT 200, Col. 69: S = Series Tracing	(See Fig. 72 for disposition of this field.)
(THIS FIELD NORMALLY FOLLOWS THE CT 2XX FOR TITLE STATEMENT, WHICH WILL BE CODED "L" OR "BLANK" IN COL. 69.)		
7. Added Entry for Partial Title	CT 20X, Col. 69: P = Partial Title	Insert "*Q" at head of string and move entire string to end of B-Fields after last CT 6XX card. Note also W-CODE needed. This "P" code is independent of whether or not main title card has "L" (for "make title added entry" in Col. 69).
8. Transliterated Title	CT 20X, Col. 69: T = Transliterated Title	Insert "!Q" at head of field and move entire string to end of B-Fields after last "*Q" or "*R" field.
9. Edition Statement	CT 20X: Scan for digit/key word occurring after end of Short Title and prior to start of CT 300.	Insert "#" in one of three positions: (1) after 4th "/", (2) after end of Rem. of Title string, (3) after end of Rem. of Title Page Transcr. Code is then followed by Edition string.
10. Imprint:		
a. Place	CT 300 = Publisher or Source	Insert 5th "/" at head of string beginning in character position 1. If data is "NP", change to

FIG. 70 (Cont.):
FORMAT TRANSLATION TABLE A-FIELD SEGMENT

NAME OF FIELD OR SUB-FIELD	SOURCE FORMAT CODE OR CONDITION	OBJECT FORMAT CODE (IN ILR INPUT FORMAT)
		"N.P." This field is always present.
b. Publisher	CT 300: Scan for punctuation (comma)	Insert 6th "/" after punctuation and prior to first character in publisher name.
c. Date of Publication	CT 300: Scan for punctuation and digits. If record contains more than one date, it will appear here in variable length format. Use as is for 7th "/". If record contains only one date, it will appear only in CT 000, Col. 31-34. Move from that position and concatenate with 7th "/".	Insert 7th "/" after punctuation and prior to first character in date (may be an alpha such as "C"). This field is always present. If data is "ND", change to "N.D." (See Fig. 71 for complex patterns)
11. Collation Statement:		
a. Pagination	CT 000, Col. 51-55: (left justified) IF REPEATED IN CT 400, IGNORE CT 000 DATA	Preface with 8th "/" and concatenate at end of data for 7th "/".
b. Illustrative Matter	CT 400	Insert 9th "/" at head of illus. matter string beginning in character position 1, and concatenate with end of pag. data moved from CT 000.
c. Size	CT 400 (This data may not have been input to the source record)	Insert 10th "/" after end of illus. matter data string, even if no "size" information is present.

FIG. 71:
DATA ELEMENT PATTERNS AND ASSOCIATED CODING IMPRINT FIELD

/PLACE,/PUBLISHER,/DATE.

$\text{/PLACE}_1\text{,\$PLACE,/PUBLISHER,/DATE.}$

$\text{/PLACE}_1\text{,/PUBLISHER}_1\text{,\$PLACE}_2\text{,\$PUBLISHER}_2\text{,/DATE.}$

/PLACE,//DATE.

> (The absence of a publisher is not signified in the data.)

/PLACE,/PUBLISHER,/DATE,DATE.

> (Two contiguous dates, separated by either a comma or a hyphen indicating a span, are not regarded as separate values of the data element for coding purposes. The two values are coded as one, in the variable field. See also I-Fields, b-codes, for fixed format dates.)

$\text{/PLACE}_1\text{,/PUBLISHER}_1\text{,1F}_{16}\text{C DATE}_1\text{;\$PLACE}_2\text{,\$PUBLISHER}_2\text{,/DATE}_2\text{.}$

> (Note the use of the MARC II delimiter preceding the first date in the example immediately above. This pattern was not foreseen in INFOCAL. The non-contiguous dates of publication are treated as repeated occurrences of the subfield and are explicitly delimited accordingly. The slash preceding the second date currently translates to a Tag 262, and will have to be re-translated to "1F$_{16}$C" in the revision of INFOCAL.)

Note that PLACE is always present: the absence of PLACE is signified by "n.p." which is to be regarded as a value of the element. Likewise, the absence of a data of publication is signified by "n.d." which is to be regarded as a value of the element.

Note also the use of the "\$" to signify repeated values of PLACE and PUBLISHER only. The translation of these "\$" is ordinally dependent on whether they follow the 5th "/" or the 6th "/". If the "\$" follows the 5th "/", it translates to Tag 260. If the "\$" follows the 6th "/", the "\$" must appear in multiples of two. The first "\$" translates to Tag 260, the second to Tag 261.

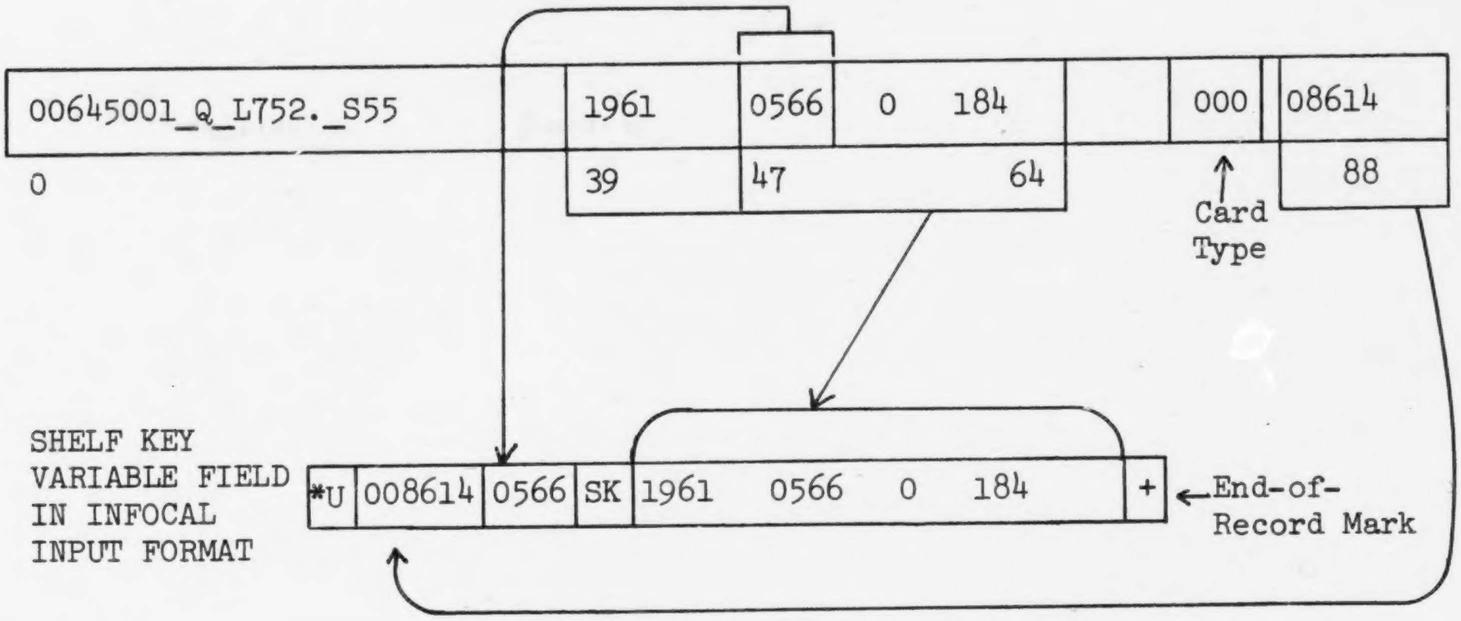
FIG. 72:
FORMAT TRANSLATION TABLE B-FIELD SEGMENT

NAME OF FIELD OR SUB-FIELD	SOURCE FORMAT CODE OR CONDITION	OBJECT FORMAT CODE (IN ILR INPUT FORMAT)
1. Bibliographic Notes: a. Series Note	CT 500: Scan for "(" in character position 1, to detect Series Note. Series string present:	See logic for Type of Added Entry Codes for Series (Fig. 68).
(1) Traced Same (Author + Title)	CT S200 = CT 500, Period detected	Insert "*A" prior to string in CT S200 and concatenate with end of 10th A-Field "/" string. Contents of CT 500 in () not carried to object record.
(2) Traced Same (Title Only)	CT S200 = CT 500 NO Period detected	Insert "*B" and proceed as above.
(3) Traced Diff.	CT S200 ≠ CT 500	Insert "*D" prior to string in CT 500 and concatenate with end of 10th A-Field string. Insert "*R" prior to string in CT S200 and concatenate after end of data from last 6XX card.
(4) Not Traced	CT 500 Series string present, but no CT S200 present	Insert "*C" prior to string in CT 500 and concatenate with end of 10th A-Field string.
SEE ALSO FIG. 69 FOR TYPE OF ADDED ENTRY CODES (W-CODES) TO BE ASSOCIATED WITH THE ABOVE SERIES NOTE PATTERNS.		
b. Other Special Notes	Data to right of ")" (close of Series Note) in CT 500 is to be scanned for key words:	
(1) Bibliography Note	"BIBLIOG"	Insert "*F" at head of string (after right paren. of Series Note, if any) and scan to end of sentence. (May be on continuation card.)

FIG. 72 (Cont):
FORMAT TRANSLATION TABLE B-FIELD SEGMENT

NAME OF FIELD OR SUB-FIELD	SOURCE FORMAT CODE OR CONDITION	OBJECT FORMAT CODE (IN ILR INPUT FORMAT)
(2) Contents Note	"CONTEN"	Insert "*I" at head of string.
c. General Notes	Else:	All other strings in CT 500 cards will be coded "*K" as the default.
2. Subject Tracings	CT 6XX, Main Headings: (1 heading = 1 card)	Insert "*M" at head of each card in character position 1.
	Subject Subdivisions: Scan for blank + hyphen as evidence of subdivision.	Expand to 2 hyphens in each case found.
3. Other Added Entries (Authors, etc.)	Each CT 1XX card beyond the first in a record; preceding card must not contain continuation sign (a minus "-" in Col. 68)	Insert "*Q" at head of each string in character position 1. Note that string may extend to more than one card.
	(See Fig. 70, paragraph 7 for disposition of title added entries in CT 2XX cards.)	
4. Shelf Key Data (for purposes of internal revision of records by Santa Cruz.) (i.e., Local System Number, Tag 035 in MARC II)	CT 000, Col. 39-64 47-50, 84-88	Insert "*U" at head of string, followed by the SC Accession Number in character positions 1-6, right justified with zero left fill, followed by Accession Date in c.p. 7-10, then letters "SK"; and move entire string intact to end of B-Fields. Concatenate with end of last "*Q", "*R" or "!Q" field in the record. (See Fig. 73) * Optional

FIG. 73:
DISPOSITION OF SHELF KEY DATA CT 000 -
SHELF KEY CARD, SANTA CRUZ


00645001_Q_L752._S55
1961
0566
0
184
000
08614
0
39
47
64
88
Card Type
SHELF KEY VARIABLE FIELD IN INFOCAL INPUT FORMAT
*U
008614
0566
SK
1961 0566 0 184
+
End-of-Record Mark

The first ten characters in the reformatted shelf key field may be used to build an index file to the master file, using the old SC accession number and the accession date as the key.

The remainder of the field, if not needed, may be dropped. Information will be extracted from Col. 39-64 to set other MARC codes, via other routines. However, it may be advisable to preserve the entire string from Col. 39-64 due to unanticipated encoding patterns in the original SC records. It is often possible to reconstruct how an error was made if the original data is easily accessible.

8. CRUNCH: THE SANTA CRUZ FILE TRANSLATION SYSTEM

By John M. Reinke

8.1 Introduction

This chapter describes the general functioning of the Santa Cruz file translation program, CRUNCH, with emphasis on the operation of the subroutines, which delimit certain types of data through the application of automatic field recognition techniques.

The purpose of this program is to accept library catalog card data coded in the UC Santa Cruz format and convert it to the ILR input format, which is considerably more complex. While there is no change of textual content, the knowledge of the nature of the content is increased by means of AFR analysis and the explicit identification which is added.

8.1.1 Main Program: LAYOUT1

The main program, called LAYOUT1, reads the set of card images (which have been punched to represent the data on a single catalog card) for a given catalog card from tape and lays out the data in core. It does some checking of the data for completeness and correctness. Particularly, it checks to insure that at least the following are included: a type 000 card (containing call number and some other information), a type 100 or type 110 card (personal or corporate author name) and a type 200 card (title). If these are not present, the data is rejected. The main program checks to see which types of data are present and which are not, and accordingly sets a series of switches or gates, which will inform the subprogram how to proceed with the processing. The subprogram STRINGER is then called. It returns the completed output string to the main program, which then writes the string out on tape, and proceeds to read in a new set of cards, etc.

8.1.2 Subprogram: STRINGER

The subprogram, called STRINGER, uses the switching information provided by the main program to set a series of gates which will insure that the data is processed in the correct manner and order, and that nothing will be erroneously excluded from or included in the final output string.

The subroutines TITLEALG and PPD will be called. Subroutines ES200500 and PERSAUTH may or may not be called, depending on the data. The subroutine TITLEALG processes the title. PPD processes the place, publisher, and date data. PERSAUTH processes the personal author data, if present. STRINGER processes the subject heading information, if present, and generates the I-field codes which correspond to this B-field information. After the data has been processed, it is all assembled into a final output string, and control is returned to the main program.

The structures of LAYOUT1 and STRINGER are large and fairly complex and are peculiar to the Santa Cruz data. The remaining subroutines may be of more general interest, since they are concerned directly with the translation functions.

Note that although the flowcharts may not represent entirely faithfully the actual coding of the subroutines, they do give a generalized outline of the logic involved, for purposes of clarity. It is expected that the comments supplied in the actual code, with the aid of the flowcharts, will permit anyone who is interested to understand the actual logic of the subroutines.

8.2 Subroutine ES200500

This subroutine compares the type 500 (comment) cards (if present) with the type S200 (serials title) cards (if also present), in order to determine the type of serials title, and to determine if the comment cards mention the presence of a bibliography, contents note, etc. An S200 card always appears together with a 500 card, but a 500 card may appear by itself. If a series title is present in the 500 data it appears at the beginning of the string and is enclosed in parentheses. The flowchart is given in Fig. 74.

First the subroutine checks to see if there is any S200 data present. If not, the subroutine checks the 500 string for left and right parentheses, which enclose the serials title if it is present. The title always appears at the start of the string. If the title is present, the parentheses are deleted and an "*c" is prefixed to the title. The remainder of the 500 string (if any) is processed by internal subroutine BIBLIO. If the title is not present, the 500 string is processed by subroutine BIBLIO to determine and delimit B-field categories.

If both S200 and 500 data are present, the length of the title in the 500 string is obtained, and the parentheses surrounding it are deleted. The lengths of the 500 title and the S200 title are compared. If they are of equal length, the titles are assumed to be identical and the S200 title is now superfluous. The title is now known to be either type "*a" (series note - traced same (author + title)) or type "*b" (series note - traced same (title only)). Internal subroutine ADDALG is then called to ascertain which type the title is, and to delimit it if it is type "*a".

Prior to the determination of the lengths of the title, the remainder of the 500 string (following the title) is processed by subroutine BIBLIO to delimit the B-field categories.

If the title lengths (as compared above) are of unequal lengths, the titles are considered to be different. The 500 title is taken as the main title and the code "*d" (series note - traced differently) is prefixed to it. The S200 title has prefixed to it the code "*r" title. The remainder of the 500 string (after the title) is processed by subroutine BIBLIO in order to determine and delimit the B-field categories.

8.3 Subroutine ADDALG

This subroutine processes a title string (see Fig. 75). The subroutine scans left to right for a period followed by two blanks. If this condition is not encountered, no delimiting is done and the title is assigned code "*b" (unless it is known in advance to be type "*r"). In either event, the delimiter "*b" is inserted after the period. The scan is continued in order to locate the last period in the string (the first period may be the last also). The portion of the title which follows the last period is now scanned for the abbreviations "v.", "vol." (for volume), and "no." (for number), and for numerals (as in "3rd" or "2nd"). If any one of these is found, the code "$1F_{16}$v" is inserted in front of it. Control is then returned to ES200500.

8.4 Subroutine BIBLIO

This subroutine deals with the remainder of the 500 string which may follow (or appear without) the 500 title (see Fig. 76). A default code of "*k" (all other notes (general)) is initially installed at the start of the string. This may be overwritten. A left to right scan is made for the terms "biblio" (for bibliography), "includes bib" and "contents." If none of these is encountered, the default code remains and control is returned to ES200500.

If either "biblio" or "includes bib" is detected, the code "*f" is prefixed to it. The address where an "*k" should now be reinserted following the words "bibliography" or "includes bibliography" is now calculated, and that address is saved. The scan now continues from that point for the word "contents." If it is not detected, an "*k" is inserted at the address just calculated. If "contents" is detected, the code "*i" is inserted in front of the word "contents." Control is then returned to ES200500.

8.5 Subroutine PPD

This subroutine deals with the place, publisher and date data which is found on type 300 and 000 cards. The flowchart is given in Fig. 77.

The subroutine first determines if there is any PPD data. If there is no data, dummy delimiters are created for what could have been the place and publisher fields. A check is made to see if there is a date on the type 000 card. If there is none, the I-field code "bn" (date not known) is assigned. If there is a date, it is examined to see whether it has four or fewer digits (as described below), and is appended to the dummy PPD string.

If there is PPD data, a left to right scan for a comma is performed on the data. If no comma is found, place only is assumed to be present. Delimiters are then inserted, and control is returned to the main routine.

If a comma is located, the routine tests to see whether it is followed immediately by a date. If it is, a delimiter (1) is inserted after the comma and before the date. This separates the place and/or publisher from the date. If the comma is not followed immediately by a date, the routine continues to scan for another comma, which is then tested for a following

---

*In the event that the word "contents" was detected during the first scan above, the code "*i" is inserted in front of it, and control is returned to ES200500.

date, and so on. If a date finally is encountered after a series of commas, the delimiter is inserted before the date, as described above. The comma which is the next to last comma is assumed to separate the place from the publisher. Therefore, a delimiter is inserted after this next to last comma. If there is no next to last comma, an additional delimiter is inserted between the place and/or publisher and the date. The place and/or publisher field is now considered by default to be the place field.

If no date is encountered after one or more commas, the last comma encountered is considered to separate the place from the publisher, and a delimiter is inserted after it. If no comma is found, place only is assumed to be present.

The subroutine has now to consider the relation of the date in the PPD string (from the type 300 card) with the date which may appear on the type 000 card. Also yet to be determined is whether there are two dates in the PPD string.

The subroutine determines whether the (first) date in the PPD string has "c" in front of it, indicating a copyright date. If there is a "c", a note is made of this fact. The subroutine next determines whether the PPD string contains a second date. If there is a second date, any date which may be present on the 000 card is ignored. If the second date has only two digits, (as in 1943-47), it is expanded to four digits by inserting the appropriate century digits. If the first date was a copyright date, it is placed second in the output string and the dates are given the I-field designation "bc" (two dates - 2d is copyright). If the first date was not a copyright date, the two dates are given the I-field designation "bm" (two dates - 2d is terminal).

If there is no second date in the PPD string, the date from the PPD string is compared with the date on the type 000 card. If the PPD date is earlier than the type 000 card date, the I-field code "br" (reprod./reprint - no digits out) is assigned.

If the PPD (type 300) date is later than the type 000 date, the I-field code is "bm" (two dates - 2d is terminal). In either case, a comma is placed at the end of the PPD string, and the type 000 date is appended to it. (Both dates will also appear in the I-field.)

If the type 300 and type 000 dates are identical, one will be examined to see if it has four digits. If it does, the I-field code "bs" (single date - no digits out) is assigned. If digits are missing, the I-field code "bq" (digits missing) is assigned, and the digits are expanded (e.g., 186- becomes two dates: 1860-1869). This date will be printed as part of the I-field code, but the original form of the PPD date will appear as such in the A-field.

In the event that there is no date, in the PPD (type 300) string, but there is a date on the type 000 card, then this date will be appended to the PPD string for printing in the A-field. The date will be expanded to four digits if need be (as described above), and in this event it will be assigned I-field code "bq". Otherwise, it will be given code "bs" (single date - no digits out).

8.6 Subroutine PERSAUTH

This subroutine deals with type 100 data (personal author names) and type 600 data (subject headings) which have been identified as personal names (see Fig. 78). Column 69 of the type 100 cards may contain a code indicating whether the person named is co-author, editor, etc.

The string is scanned left to right for the appearance of the word "Sir" (the most common title of honor) in the string. If it is present, it is removed from the string, and a note made of this fact. The string then is scanned for the presence of a date. If a date is found, the delimiter "$b" is inserted in front of it. The delimiter "$a" is inserted at the head of the string. If the word "sir" was previously encountered, it is inserted back into the string, after the name and before the date. The delimiter "$c" is inserted in front of the word "sir." If no date was encountered, the word "sir" with its delimiter will still be appended to the end of the name.

The code in column 69 of the type 100 card is examined. If the code is present, the appropriate abbreviation with the delimiter "$e" is attached to the end of the type 100 string. For instance, the code "c" in column 69 stands for "compiler." Therefore, when this code is detected, the substring "$ecomp." will be appended to the name string.

8.7 Subroutine TITLEALG

This subroutine delimits the subfields of the monograph title, which appears on the SC type 200 cards (see Fig. 79).

a. The subfields which may be present are (in order): (1) short title (always present); (2) long title; (3) author statement and additional information; (4) edition statement; (5) remainder of edition statement. The short title is the third A-field, according to the ILR input format. Subfields (2) through (5), above, constitute (when present) the fourth A-field.

b. The title string is scanned initially for the following: (1) a question mark or period followed by two blanks, which would be taken as end of sentence indicators; (2) a colon or semi-colon, which would be taken as end of short title indicators; (3) the word "by," (either standing alone or enclosed in brackets), which would be taken as an end of short title indicator.

c. If a question mark or period followed by two blanks is encountered, everything preceding this punctuation symbol is taken as belonging to the short title subfield. If there is no further data, control is returned to the subprogram STRINGER.

d. The routine next tests to determine if the remainder of the string begins with a numeral or the characters "Rev" (for "Revised" or "Rev."). The presence of any of these is taken to indicate the start of the fourth subfield, the edition statement.

e. The second and third subfields are considered absent. The appropriate delimiters are inserted, and this remainder string is then scanned for the word "by." If "by" is present, it is assumed to mark the beginning of the fifth subfield (the remainder of the edition statement), and is so marked. Control is returned to the subprogram.

f. If the remainder of the string does not start with numerals or "Rev," it is considered to constitute only the third subfield (author statement and additional information). It is so marked (unless it already has been marked; see H. below), and control is returned to the subprogram.

g. If, during the initial scan (B.), a colon or semi-colon is encountered, this is taken to indicate the end of the short title. What follows immediately is considered to be the long title (or title elaboration - second subfield). This remainder is then scanned for a period followed by two blanks or the word "by" (or "[by]").

h. The word "by" or ("[by]") is taken to delimit the start of the third subfield, and the appropriate delimiter is inserted if it is encountered. In the event it is encountered, the remainder of the string is scanned for a period followed by two blanks. If no period is found, the remainder of the string thus scanned is considered to be a continuation of the third subfield and no further delimiters are inserted; control is returned to the subprogram. If a period followed by two blanks is found, control (for scanning remainder of string) is transferred to that portion of the program outlined in paragraphs D. to F.

i. If, during the second scan (described in paragraph G.), a period followed by two blanks is encountered, the period is considered to signal the end of the second subfield (long title or title elaboration). Control (for scanning the remainder of the string) is transferred to that portion of the program outlined in paragraphs D. to F.

j. If, during the initial scan (described in paragraph B.), the word "by" or ("[by]") is found, it is taken to delimit the first and third subfields, with the second subfield considered to be absent. The remainder of the string is scanned for a period followed by two blanks. If none is found, the remainder is considered to belong entirely to the third subfield, and control is returned to the subprogram. If a period followed by two blanks is found, control (for scanning the remainder of the string) is transferred to that portion of the program outlined in paragraphs D. to F.

k. If, during the initial scan (described in paragraph B.), none of the scan characters or words is found, the title is considered to consist only of the short title. Delimiters are accordingly inserted and control is returned to the subprogram.

FIG. 74: FLOWCHART OF SUBROUTINE ES200500

This subroutine compares the SC type 500 (comment) cards (if present) with the SC type S200 (serials title) cards (if also present), in order to determine the type of serials title, and to determine if the comment cards mention the presence of a bibliography, contents note, etc. An S200 card always appears together with a 500 card, but a 500 card may appear by itself. If a series title is present in the 500 data, it appears at the beginning of the string and is enclosed in parentheses.

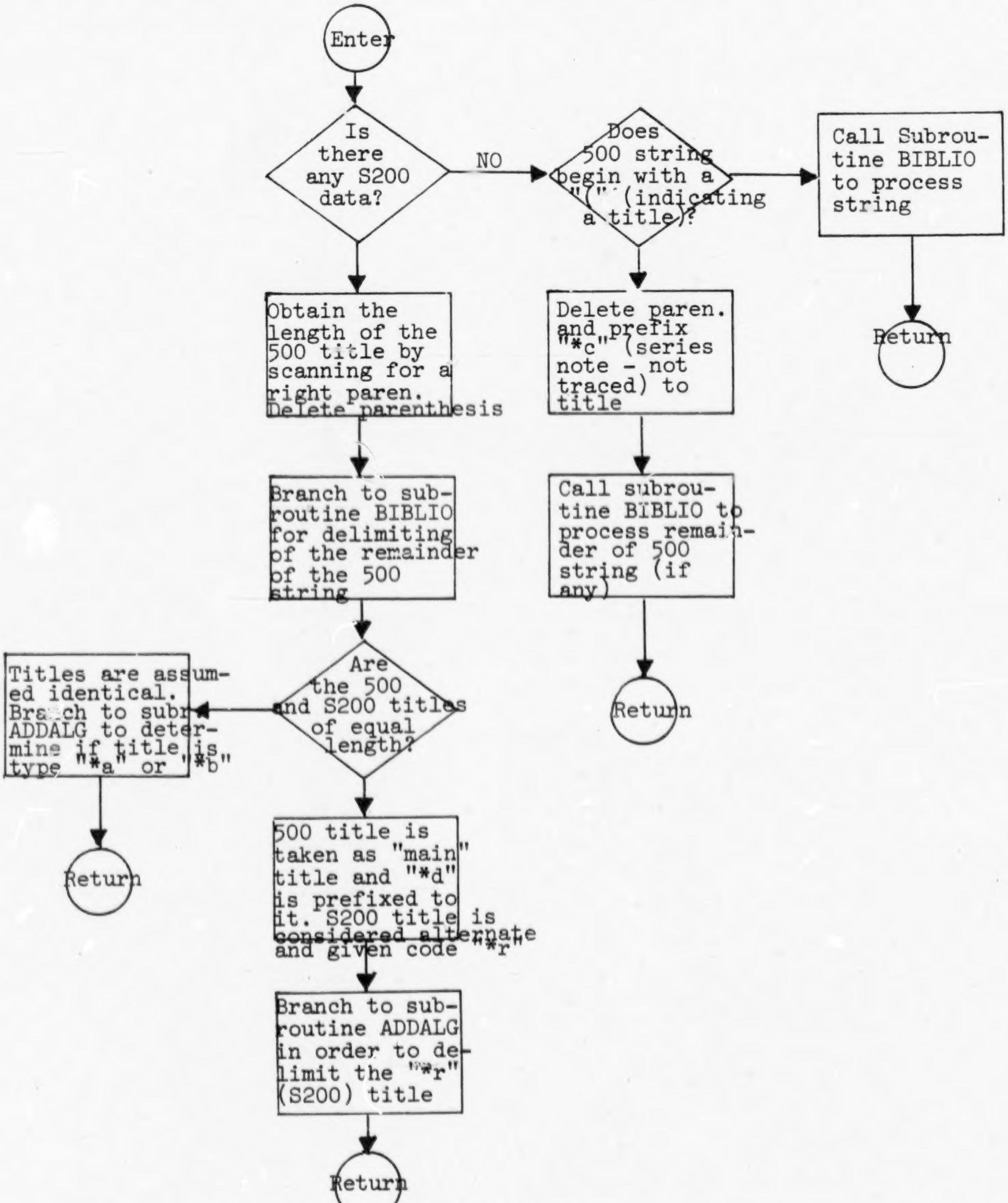

Enter
Is there any S200 data?
NO
Does 500 string begin with a "(" (indicating a title)?
Call Subroutine BIBLIO to process string
Return
Obtain the length of the 500 title by scanning for a right paren. Delete parenthesis
Delete paren. and prefix "*c" (series note - not traced) to title
Branch to subroutine BIBLIO for delimiting of the remainder of the 500 string
Call subroutine BIBLIO to process remainder of 500 string (if any)
Titles are assumed identical. Branch to subr ADDALG to determine if title is type "*a" or "*b"
Are the 500 and S200 titles of equal length?
Return
Return
500 title is taken as "main" title and "*d" is prefixed to it. S200 title is considered alternate and given code "*r"
Branch to subroutine ADDALG in order to delimit the "*r" (S200) title
Return

FIG. 75: FLOWCHART OF INTERNAL SUBROUTINE ADDALG

This subroutine (internal to subroutine ES200500) determines whether a title is of type "*a" (series note traced same (author + title)) or of type "*b" (series note traced same (title only)). A code will be returned to ES200500 to indicate which type the title is. This will be disregarded in the event that ES200500 has determined previously that the title is type "*r". If the title is type "*a", it will be delimited.

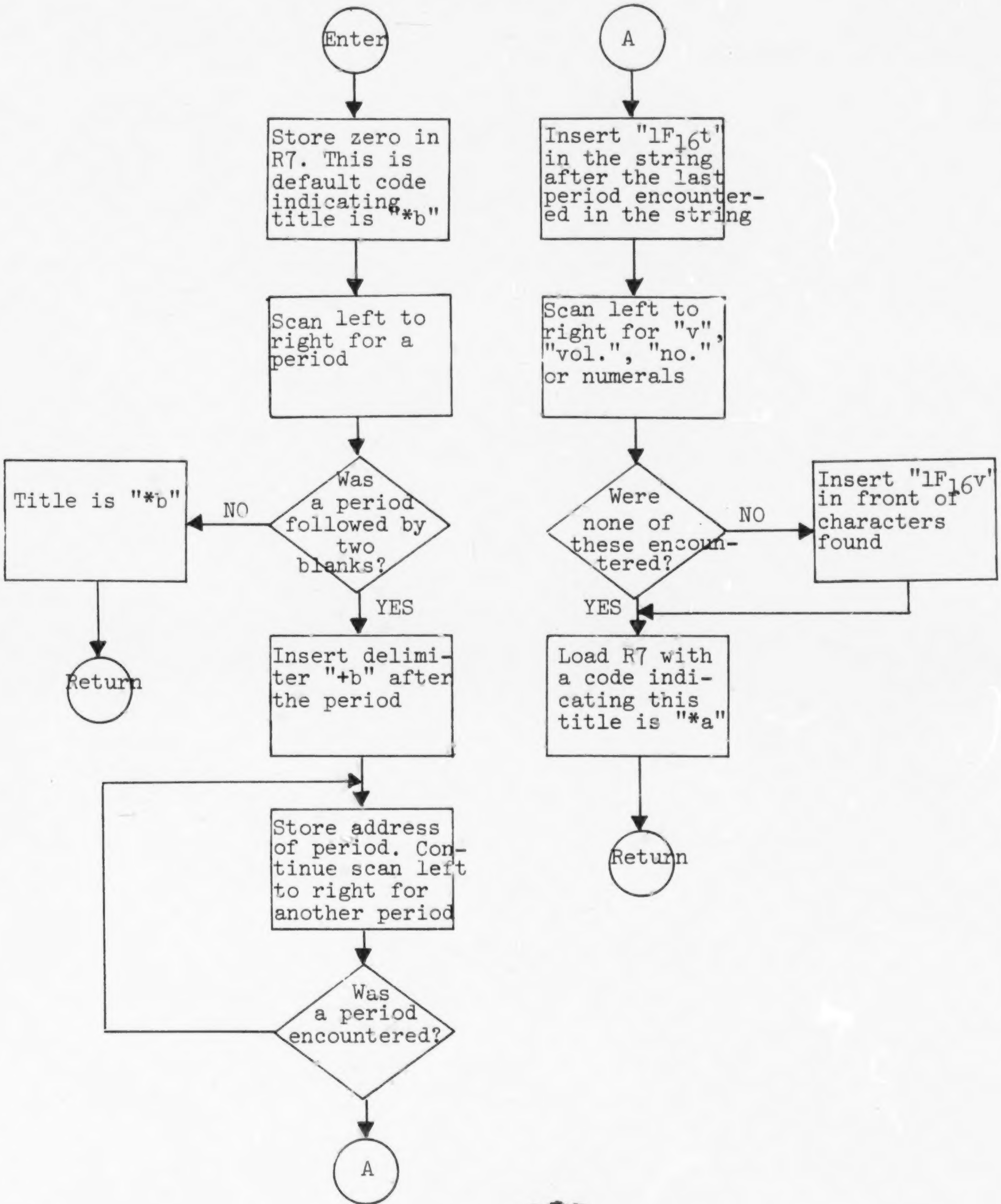

Enter
Store zero in R7. This is default code indicating title is "*b"
Scan left to right for a period
Was a period followed by two blanks?
NO
Title is "*b"
Return
YES
Insert delimiter "+b" after the period
Store address of period. Continue scan left to right for another period
Was a period encountered?
A
A
Insert "1F16t" in the string after the last period encountered in the string
Scan left to right for "v", "vol.", "no." or numerals
Were none of these encountered?
NO
Insert "1F16v" in front of characters found
YES
Load R7 with a code indicating this title is "*a"
Return

FIG. 76: FLOWCHART OF INTERNAL SUBROUTINE BIBLIO

This subroutine (internal to subroutine ES200500) assigns the B-field category indicator "*k" to those portions of the commentary data which cannot be identified as serial title, bibliographic information (code "*f"), or contents information (code "*i").

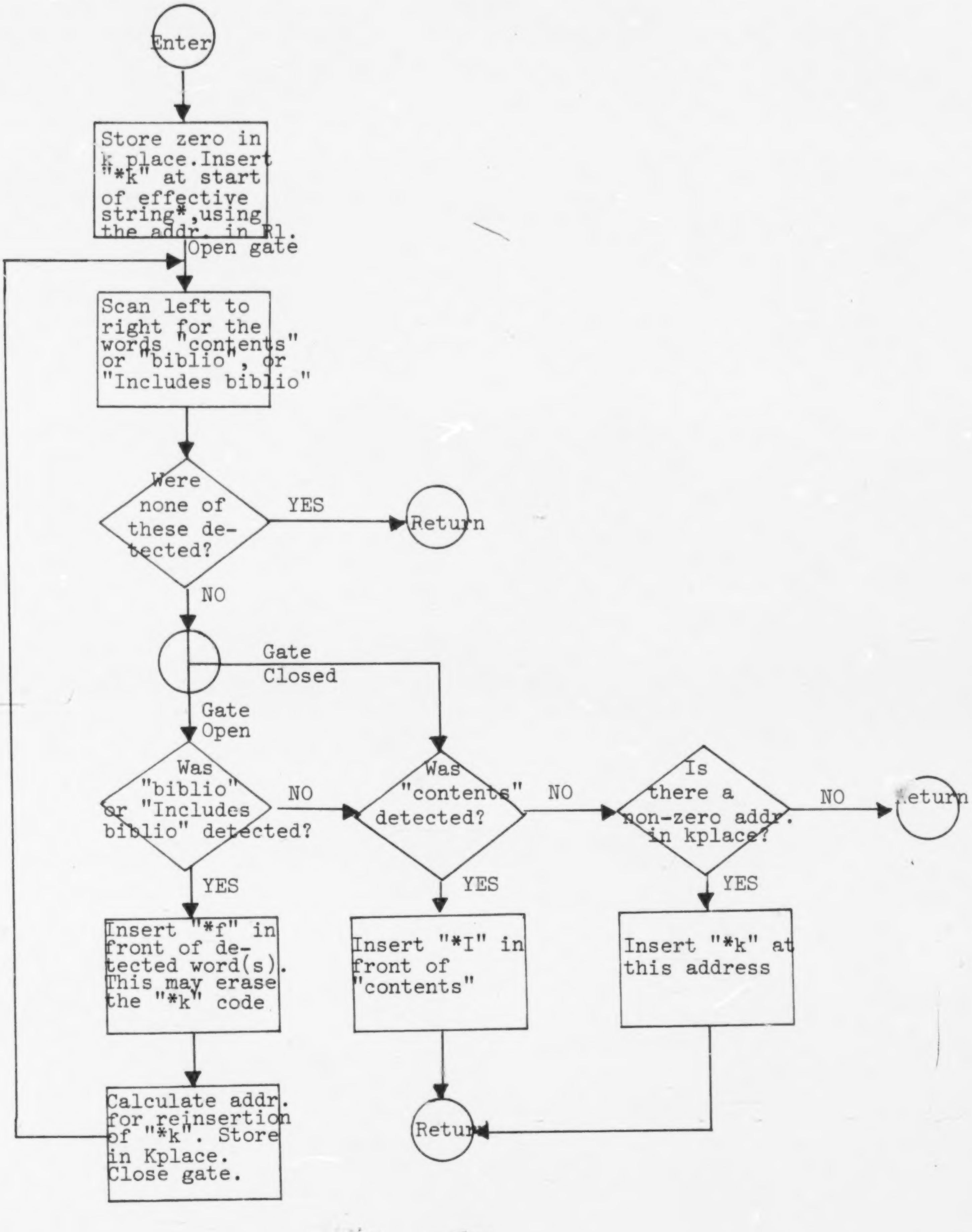

Enter
Store zero in k place.Insert "*k" at start of effective string*,using the addr. in R1. Open gate
Scan left to right for the words "contents" or "biblio", or "Includes biblio"
Were none of these detected?
YES
Return
NO
Gate Closed
Gate Open
Was "biblio" or "Includes biblio" detected?
NO
Was "contents" detected?
NO
Is there a non-zero addr. in kplace?
NO
Return
YES
Insert "*f" in front of detected word(s). This may erase the "*k" code
YES
Insert "*I" in front of "contents"
YES
Insert "*k" at this address
Calculate addr. for reinsertion of "*k". Store in Kplace. Close gate.
Return

FIG. 77: SUBROUTINE PPD FLOWCHART

This flowchart shows the conceptual logic by which subroutine PPD handles the place, publisher, and date data which is found on SC card types 300 and 000 (date only).

Aim: (1) to create the I-field date and to ascertain the I-field date type; and (2) to create the A-fields for place, publisher and date by delimiting the 300 string and making use of the date supplied on the 000 card.

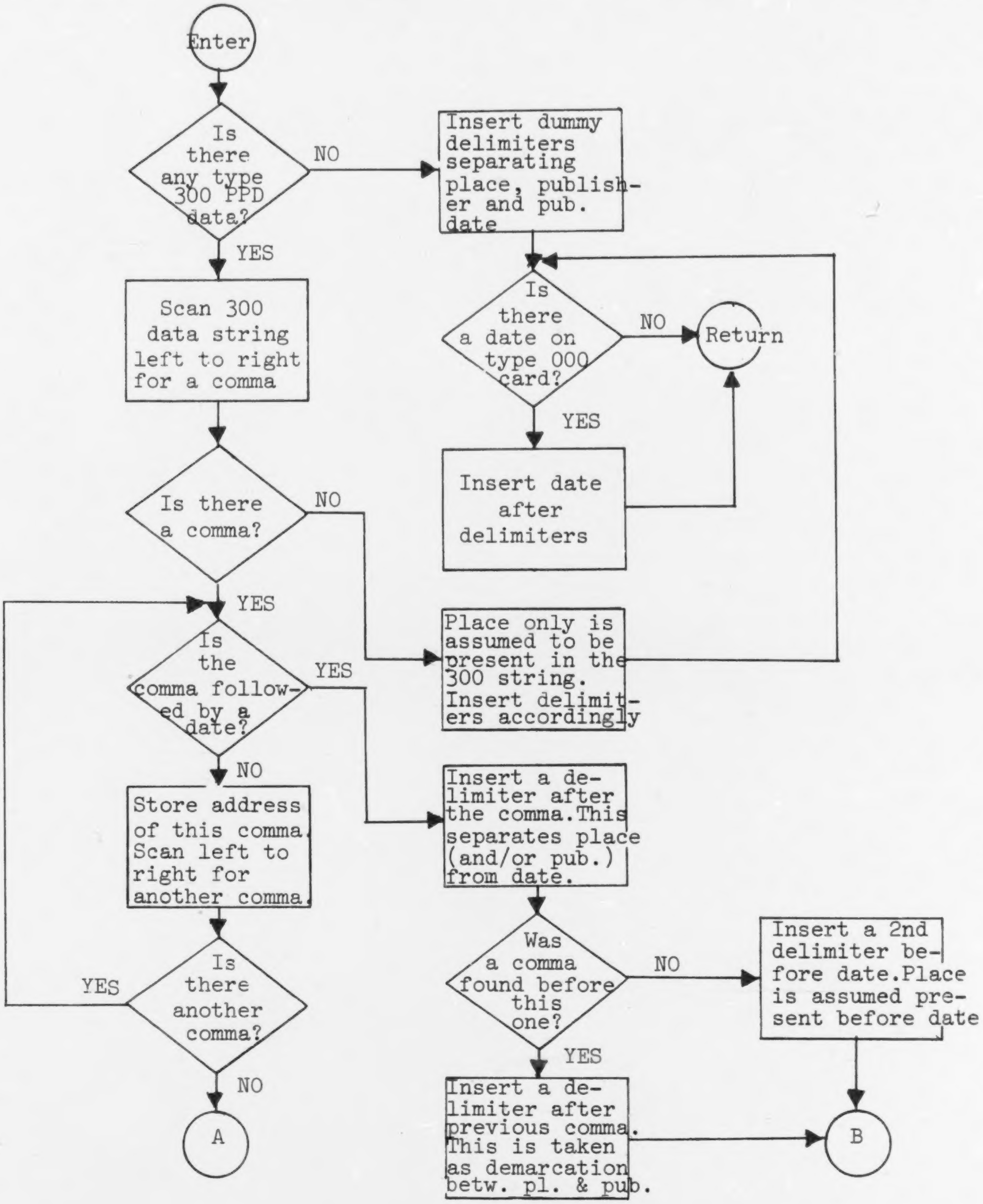

Enter
Is there any type 300 PPD data?
NO
Insert dummy delimiters separating place, publisher and pub. date
YES
Scan 300 data string left to right for a comma
Is there a date on type 000 card?
NO
Return
YES
Insert date after delimiters
Is there a comma?
NO
YES
Place only is assumed to be present in the 300 string. Insert delimiters accordingly
Is the comma followed by a date?
YES
NO
Store address of this comma. Scan left to right for another comma.
Insert a delimiter after the comma. This separates place (and/or pub.) from date.
Is there another comma?
YES
NO
A
Was a comma found before this one?
NO
Insert a 2nd delimiter before date. Place is assumed present before date
YES
Insert a delimiter after previous comma. This is taken as demarcation betw. pl. & pub.
B

FIG. 77: FLOWCHART OF SUBROUTINE PPD (Cont.)

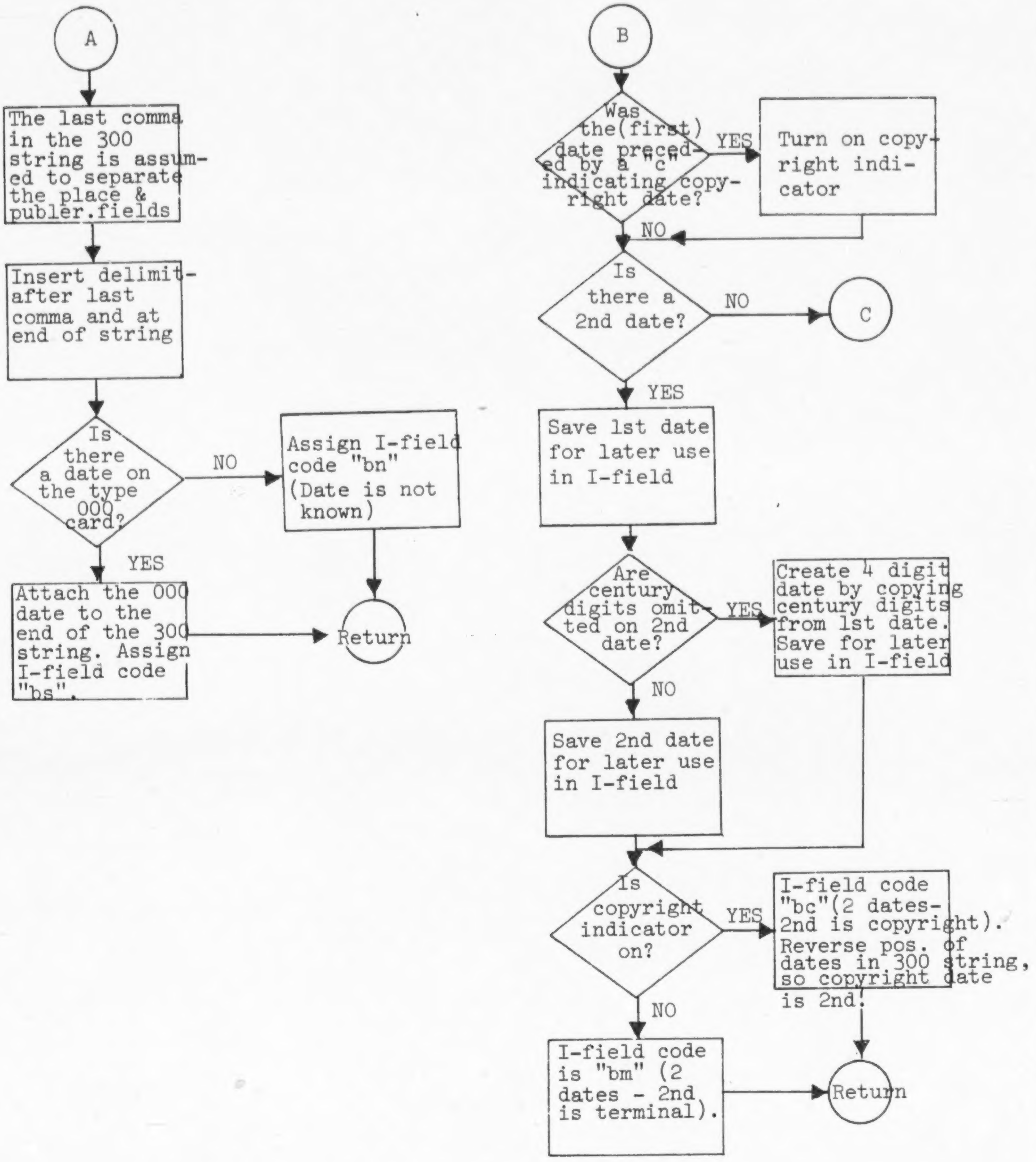

A
The last comma in the 300 string is assumed to separate the place & publer.fields
Insert delimit-after last comma and at end of string
Is there a date on the type 000 card?
NO
Assign I-field code "bn" (Date is not known)
YES
Attach the 000 date to the end of the 300 string. Assign I-field code "bs".
Return
B
Was the(first) date preceded by a "c" indicating copyright date?
YES
Turn on copyright indicator
NO
Is there a 2nd date?
NO
C
YES
Save 1st date for later use in I-field
Are century digits omitted on 2nd date?
YES
Create 4 digit date by copying century digits from 1st date. Save for later use in I-field
NO
Save 2nd date for later use in I-field
Is copyright indicator on?
YES
I-field code "bc"(2 dates- 2nd is copyright). Reverse pos. of dates in 300 string, so copyright date is 2nd.
NO
I-field code is "bm" (2 dates - 2nd is terminal).
Return

FIG. 77: FLOWCHART OF SUBROUTINE PPD (Cont.)

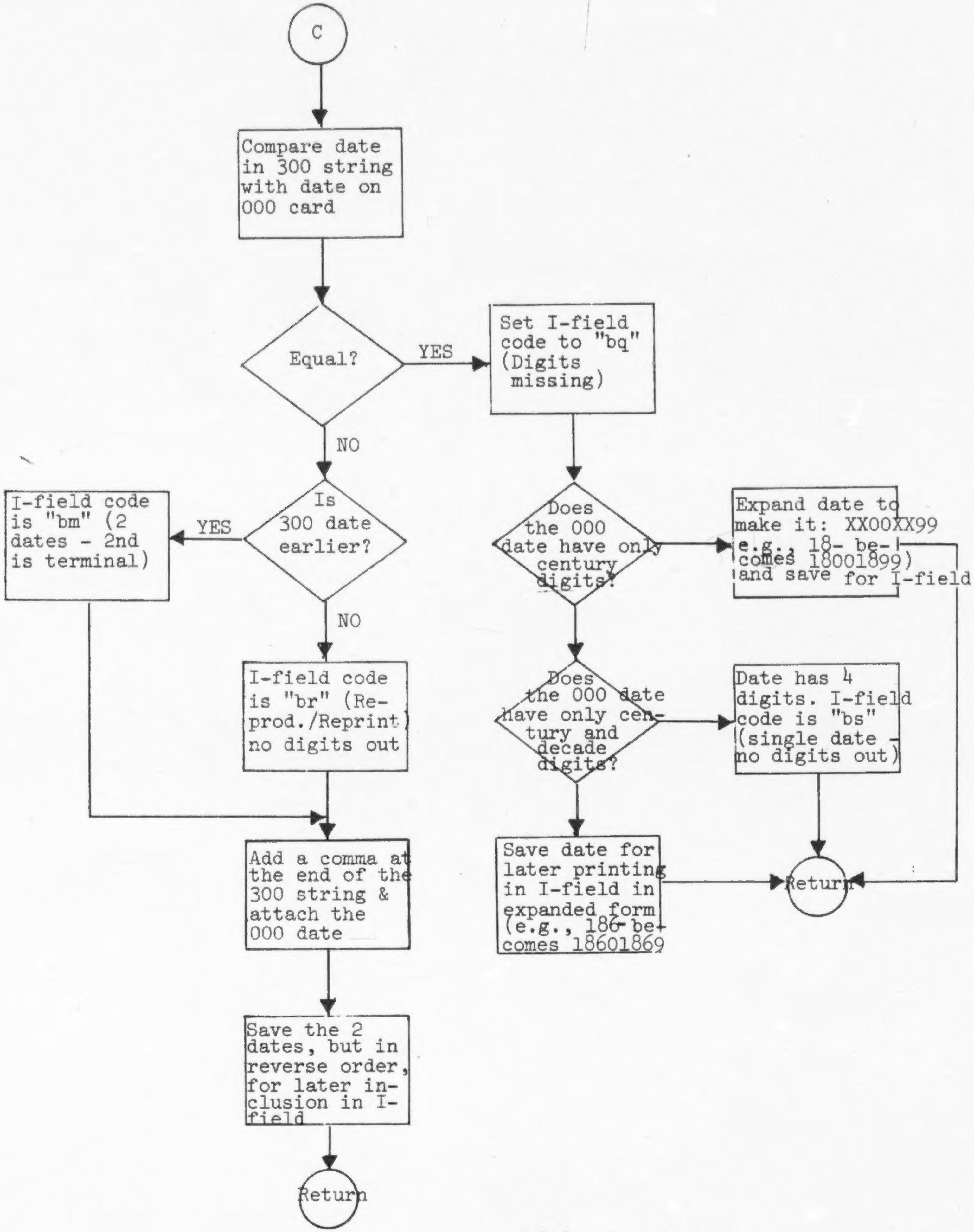

C
Compare date in 300 string with date on 000 card
Equal?
YES
Set I-field code to "bq" (Digits missing)
NO
Is 300 date earlier?
YES
I-field code is "bm" (2 dates - 2nd is terminal)
NO
I-field code is "br" (Reprod./Reprint) no digits out
Add a comma at the end of the 300 string & attach the 000 date
Save the 2 dates, but in reverse order, for later inclusion in I-field
Return
Does the 000 date have only century digits?
Expand date to make it: XX00XX99 (e.g., 18- becomes 18001899) and save for I-field
Does the 000 date have only century and decade digits?
Date has 4 digits. I-field code is "bs" (single date - no digits out)
Save date for later printing in I-field in expanded form (e.g., 186- becomes 18601869
Return

FIG. 78: FLOWCHART OF SUBROUTINE PERSAUTH

This subroutine scans the personal author data string (SC card type 100) and delimits the name subfield ("$1F_{16}$a"), together with the date ("$1F_{16}$d"), title of honor (i.e., "sir" - "$1F_{16}$c"), and relator (e.g., "editor" - "$1F_{16}$e") if any or all of these is present.

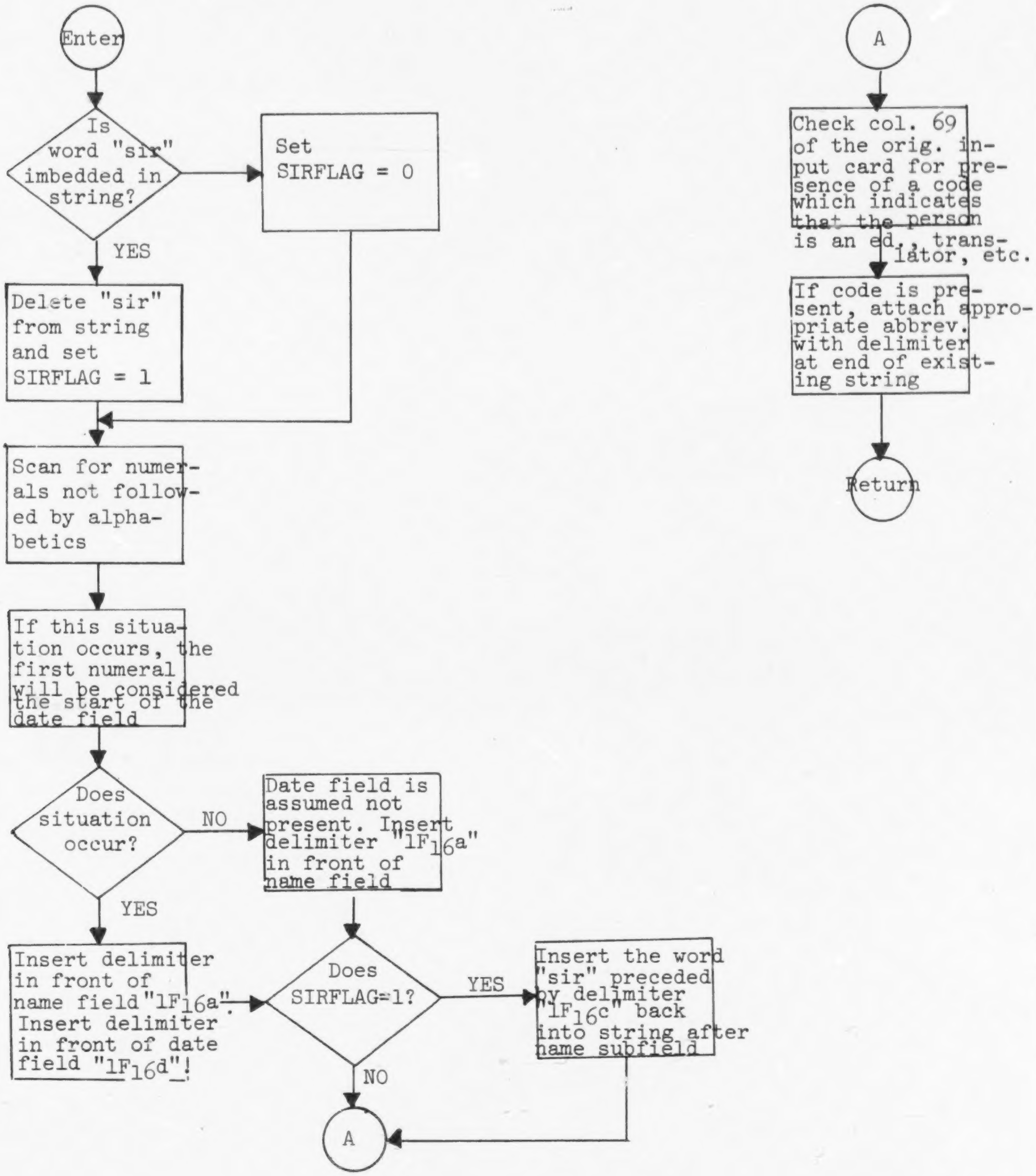
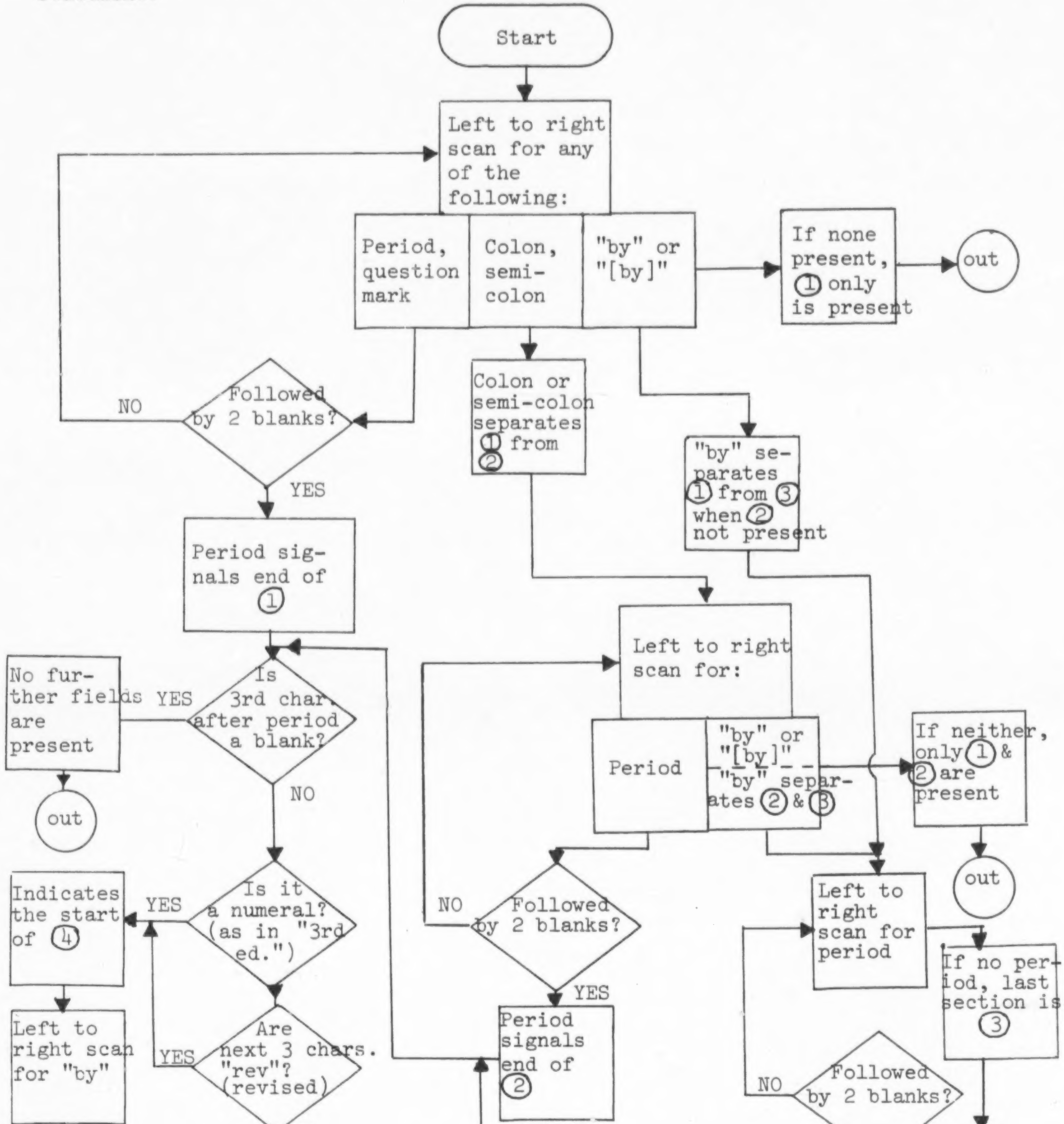

Enter
Is word "sir" imbedded in string?
Set SIRFLAG = 0
YES
Delete "sir" from string and set SIRFLAG = 1
Scan for numerals not followed by alphabetics
If this situation occurs, the first numeral will be considered the start of the date field
Does situation occur?
NO
Date field is assumed not present. Insert delimiter "$1F_{16}$a" in front of name field
YES
Insert delimiter in front of name field "$1F_{16}$a". Insert delimiter in front of date field "$1F_{16}$d"!
Does SIRFLAG=1?
YES
Insert the word "sir" preceded by delimiter "$1F_{16}$c" back into string after name subfield
NO
A
A
Check col. 69 of the orig. input card for presence of a code which indicates that the person is an ed., translator, etc.
If code is present, attach appropriate abbrev. with delimiter at end of existing string
Return

FIG. 79: SUBROUTINE TITLEALG FLOWCHART

Given: Complete monograph title.
To be determined: (1) short title, (2) long title, (3) author statement and additional information, (4) edition statement, (5) remainder of edition statement.



Start
Left to right scan for any of the following:
Period, question mark
Colon, semi-colon
"by" or "[by]"
If none present, ① only is present
out
Followed by 2 blanks?
NO
YES
Colon or semi-colon separates ① from ②
"by" separates ① from ③ when ② not present
Period signals end of ①
Left to right scan for:
Period
"by" or "[by]"
"by" separates ② & ③
If neither, only ① & ② are present
out
No further fields are present
YES
Is 3rd char. after period a blank?
NO
out
Indicates the start of ④
YES
Is it a numeral? (as in "3rd ed.")
NO
Followed by 2 blanks?
YES
Period signals end of ②
Left to right scan for period
If no period, last section is ③
out
Left to right scan for "by"
YES
Are next 3 chars. "rev"? (revised)
NO
NO
Followed by 2 blanks?
YES
If "by" is present, it separates ④ from ⑤
This is either the begin. or cont. of ③ to end of string
out

9. PROSPECTS FOR AUTOMATIC FIELD RECOGNITION

By John M. Reinke

9.1 Introduction

At present, automatic field recognition (AFR) is an "art," not a "science." The development of algorithms seems to depend on the intuitive recognition of patterns of content-structure associations in the data by the analyst. Hence, the success of the algorithms hinges on the presence or absence of certain punctuation keys and a few keywords, such as "by," "edited," etc. Algorithms are developed by an initial "eyeballing" of a sample of the data base. Output is then checked against a hand-corrected set of the same data, a feedback process then ensues through which the algorithms are further refined, and more marginal data cases subsequently are correctly processed. This process increases the complexity of the algorithm and usually results in some sacrifice of speed for greater accuracy.

Existing algorithms make relatively little use of pre-scanning or "look-ahead" techniques, but charge through the data-strings head-on, after the fashion of a python swallowing a cow. Consequently, existing algorithms take a shotgun approach to the data, being designed very generally to handle any extreme data case of the type under consideration. In contrast to an approach of matching the rifle calibre to the type of game being hunted, this approach presupposes that the type of game being stalked has been more or less successfully identified in advance.

However, it seems entirely possible to create a standardized and systematic approach to algorithm design, through a careful analysis of what actually transpires (in the analyst's mind) in the current "subjective" approach and through the introduction of new analytic techniques, such as pre-scanning.

It is possible to foresee the development of a computer language which will both facilitate algorithm design and enable the analyst quickly to classify and quantify his data in such ways as to make explicit the patterns which are implicit in the data. Constraint parameters for speed and accuracy, together with optional methods of attack will be available to the user. Newly developing data compression and indexing schemes, coupled with the steady increase in available computational power (and decrease in cost) will make possible any desired degree of algorithmic accuracy, through a change in the content-to-structure mix of the input data. That is, it will be possible by explicit enumeration to treat former content matter (such as given words and phrases) as structure. (The idea of structure vs. content needs to be clarified. It has to do with the computer's ability to make pattern recognitions but not associative recognitions.)

In the short run, an analysis by which analysts conceive and execute algorithms should result in some "algorithms" which will not only facilitate the teaching of algorithm design to others, but also speed up the process and increase the accuracy of the end product. As we learn more about the process, we will be able to produce programs to aid the algorithm design process by applying classification and pattern-emergence techniques to the sample data bases under consideration. At this stage the algorithm design process will have been partially automated, with the transference of lower levels of data analysis from the analyst to the computer.

9.2 An Example of the Proposed Use of Pre-scanning in Algorithm Design

Assume that the input data is the imprint field of a catalog card. It is desired to design an algorithm which will identify the place, publisher, and date elements within this field.

These elements can appear in at least the following combinations:

Class	Type
1	PLACE, PUBLISHER, DATE.
2	PLACE$_1$, PLACE$_2$, PUBLISHER, DATE.
3	PLACE$_1$, continuation of PLACE$_1$, PUBLISHER, DATE.
4	PLACE$_1$, PUBLISHER$_1$, PLACE$_2$, PUBLISHER$_2$, DATE.
5	PLACE, DATE.
6	PLACE, PUBLISHER, DATE, DATE.
7	PLACE$_1$, PUBLISHER$_1$, DATE$_1$; PLACE$_2$, PUBLISHER$_2$, DATE$_2$.
8	PLACE, PUBLISHER.

It is not necessary to have decided beforehand what the various combinations are or may be. However, it is necessary to make an initial decision as to which structural elements will delimit or identify the content elements in the imprint field. Suppose that the key structural elements are taken to be the comma, period, and semi-colon (,.;).

We now can write a pre-scan routine which will process some reasonable samples of imprint fields and produce a characteristic "key" for each imprint field. The type classes of imprint fields listed above would have the following keys:

Type Class	Key	Key Class
1	',,.'	1
2	',,,.'	2
3	',,,.'	2
4	',,,,.'	3
5	',.'	4
6	',,,.'	2
7	',,;,,.'	5
8	',.'	4

Note that only five unique keys have been produced using the given structural element set. If the ten digits are added to the existing structural element set, the following keys will be generated:

Type Class	Key	Key Class
1	',,9.'	1
2	',,,9.'	2
3	',,,9.'	2
4	',,,,9.'	3
5	',9.'	4
6	',,9,9.'	5
7	',,9;,,9.'	6
8	',.'	7 where 9 stands for any digit

(The possibilities now become apparent for the development of efficiency equations for a given structural element set.) The keys are composed of various members of the structural element set. Observe that there are now seven unique keys for the eight type classes.

The pre-scan routine will operate in the following manner: after the analyst decides upon the structural element set, to be input to the program, the program scans an input imprint field and produces its characteristic key. It then compares that key with unique key patterns which have been stored already, and either assigns the input imprint field a number corresponding to the key on which a match was obtained, or if it is unique, adds the current key to the table and assigns the corresponding number to the input imprint field. The input field with its newly assigned number will then be written out on tape.

After all input data has been processed, the table of unique keys will be dumped, and the output tape will be sorted so that the imprint fields will be grouped according to their characteristic key patterns. These will then be printed.

The output of this program will enable the analyst to ascertain quickly both the most common type class (or classes) of data in the imprint field, and the degree of consistency (the presence of more than one type class within a given key class) within a given key class for a given structural element set. Output from different structural element sets can be compared.

This data will provide the analyst with a much clearer picture of the accuracy he can expect than he could obtain merely from "eyeballing" the data before designing an algorithm.

In fact, it will now be possible for the analyst to design several algorithms, perhaps one for each characteristic key. During actual processing, pre-scanning can be used to produce a key which will then route the imprint field to the specific algorithmic subroutine designed for it. Hence, what formerly would have been tackled by one subroutine will now be accomplished instead by a whole family of subroutines.

9.3 A Proposed System for AFR Algorithm Development

It is possible to establish an interactive system for the development and modification of algorithmic field recognition routines (to achieve optimum data transformation) through a continual monitoring of the data during retrospective conversion of large files. The proposed system will operate in the following way:

a. An analyst seated at a CRT terminal will ask for a relatively small random sample n (of the data for which an AFR algorithm is to be developed) to be flashed on the screen from files stored on disk or tape.

b. The analyst will choose an initial set of keys on the basis of his (brief) analysis of the data.

c. These keys will be input to the pre-scan program (the nature of which will be described below) which will be run against another random sample m (>n) of a suitable size.

d. The results of this program will be flashed on the screen for further analysis (if necessary) by the analyst. The results will be of the following sort:

    1. A list of the k key classes generated from the sample m.

    2. The elements of sample m will be sorted into k groups according to the key class to which they belong, and the various frequencies (F_1 to F_k) of the elements of m belonging to the k key classes will be calculated. (These will be calculated as percentages. Example: suppose $m = 100$ and $k = 5$. If 25 elements of m belong to the second key class, then $F_2 = 25\%$.

e. The analyst will examine the k groups of the elements of the sample m for internal consistency (i.e., is there more than one type class within a given key class?), depending on their associated frequencies. For example, if a given group k_i has a frequency $F_i < 2\%$, the analyst may not wish to determine the internal consistency of the group, because it appears (for his purposes) so rarely.

    The analyst also will determine whether the original set of keys generate an inordinate number of spurious key classes (the case of a single type class generating a number of key classes).

f. On the basis of this analysis, the analyst will have at least four options:

   1. The analyst may decide that the set of keys has produced key classes of such consistency and favorable frequency that no further pre-scanning or modification of the key set is necessary. As an example, suppose a key set has generated 5 key classes with frequencies $F_1 = 25\%$, $F_2 - 20\%$, $F_3 = 15\%$, $F_4 = 38\%$, $F_5 = 2\%$. The analyst has determined that key classes k_1 to k_4 are internally consistent, but that k_5 is very inconsistent. He may decide in this case that the 2% of cases represented by k_5 are negligible.

   2. The analyst may decide to delete or add to the set of keys and repeat steps C-F in order to generate new key classes, which hopefully will be more internally consistent than the old key classes. Efficiency functions can be developed which will give the analyst another criterion for deciding which of two keys to employ in creating his algorithms. For example, suppose that one set of three keys generates 6 key classes with a net inconsistency of 3%, while another set of 5 keys generates 12 key classes with a net inconsistency of 2-1/2%. Efficiency functions will help the analyst decide which set to use.

   3. The analyst may decide upon a secondary set of keys which will be used to reprocess those original (primary) key classes which have exhibited internal inconsistency. The hope is that the secondary key classes thus produced will be internally consistent. The analyst will then repeat steps C-F. This approach is an alternative to step F_2. Again, it will be possible to develop criteria for determining whether approach F_2 or F_3 is superior in a given situation.

   4. The analyst may decide that the results of the prescan are conditionally acceptable. For example, he may find that 90% of the members of sample m belong to key classes which are internally consistent. The other 10% may belong to a key class k_i which contains two type classes. Suppose that 70% of the members of key class k_i belong to one type class and that the other 30% of the members belong to the other type class. The analyst may find it acceptable to specify a default algorithm for the 70% and an alternate algorithm for the 30%. The use of the alternate algorithm may occur either during post-editing or during actual processing. Its implementation will be discussed below.

g. If the analyst has decided on either step F_1 or F_4, he may wish to check the generality of his finding by applying his chosen key set (s) against one or two other random samples drawn from the data file.

h. Assuming the successful completion of step G, the analyst is ready to write some algorithmic sentences which will be translated (probably by a SNOBOL-type program) into a series of logical operators (notation for which must be developed or borrowed from somewhere).

   These operators together with some output from the pre-scan program (such as the final key set, key classes and some mapping tables to be explained later) will be input to another program (language to be determined) which will generate either assembly language or machine language code. This final code will be used actually to process the data.

   As an example, the following could be the sort of algorithmic statement which might be written to generate the appropriate code for sub-fielding type class 1, which has the form: "PLACE, PUBLISHER, DATE." and which has the key class:',,.' "Key class 1: insert "1" after the first comma and '$c' after the second comma."

i. With this type of system, it is reasonable to expect that in many cases an analyst will be able to produce executable code (for a given class of data) in a morning or day's time.

9.4 Advantages of the AFR Algorithm Development (AFRAD) System

a. It will produce results much more rapidly than they can be obtained with existing methods.

b. It will produce a family of simple (default and alternate) algorithms to replace the single complex algorithm produced at present. Since the algorithms are simpler, they will be much easier to write or modify. Due to their simplicity, they may be written by persons who are relatively unfamiliar with algorithmic techniques.

c. This method potentially is more accurate than the current method.

   During the processing of large files, this approach will make possible the maintenance of a running statistical check of the frequency with which certain (key) classes are appearing. It will then be possible (if necessary) to modify certain algorithms using cumulative occurrence data to improve on the design originally formulated on the basis of limited sample data.

e. Since the algorithms are produced by writing algorithmic statements, anyone who wishes to understand the functioning of a given algorithm can do so by reading the algorithmic statements. In effect, the algorithmic statements provide flow charts of the functioning of the algorithms.

f. It will be possible during processing to tag, for post-editing, those (key) classes of data which the AFRAD system has indicated will be most inconsistent.

9.5 Remarks on Sampling

The algorithms to be developed in the AFRAD system will be based on patterns which occur in the random sample(s) drawn from the data file. It is well to remember that the frequencies obtained from the sample are only estimates of the true population frequencies. It will be impossible to predict whether the actual frequencies will be more favorable or less favorable to the correct conversion of the data. This is important to recall in connection with those key classes which exhibit some degree of internal inconsistency: they may comprise a larger section of the file than the sample would lead you to believe. Some key classes may be so rare as not to appear in the sample at all.

However, the processing program will be able to detect them. These can be dealt with during processing by saving the numbers of the records in which they appear, and saving the actual examples until some specified number has been accumulated. These will then be read out. An analyst can write an algorithm to handle them which will be incorporated into the processing program. Those records already processed can be re-processed also.

A "run" is a phenomenon of some interest in probability theory. An example of a "run" could be having a tossed coin come up heads 50 times in a row. We might call a "near run" the case where the coin comes up heads 45 times in a total of 50 times. These phenomena may have some bearing on our work.

Suppose we had a million marbles, 70% of which were black and 30% white. Assume that they were mixed about (integrated) and then lined up in a single row (or string). According to probability theory, it is highly unlikely that the white marbles would everywhere be regularly distributed among the black marbles. There probably would be a fair amount of "clumping" of both the white marbles and the black marbles. That is, it would be possible to pick out some segments of the string which were composed of all, or nearly all white marbles, even though the predominant color is black. Using a continuous left-to-right sampling technique, taking every nth marble, we could probably predict with a reasonable degree of accuracy when a given segment (of length m marbles) of the string contained a percentage of black marbles equal to p (some arbitrarily chosen percentage) or greater.

The application to our work is as follows: Suppose that a certain key class comprises 35% of a given class of data. Suppose also that the key class contains two type classes (analogous to the black and white marbles) which cannot be distinguished readily from each other by punctuation keyword lists, or the like. It would be possible during processing to use the sampling technique described to determine (with some reasonable degree of accuracy) whether a given section of the data contained mostly one type class or the other. Based on this, the appropriate default or alternate algorithm could be invoked. The problem is that sample taking and analysis is likely to be much slower than the speed of the program processing the data, because some form of human intervention will be required to determine the type class.

If this problem should prove insurmountable, it might be better to apply this technique at the post-editing stage (or maybe forget about it altogether). The advantage of this approach (if implementable) is that it produces a file with a lower absolute amount of error. The disadvantage is that now, instead of one type of error (some A's should be B's), there are two types of errors (some A's should be B's, and some B's should be A's). This approach will be most successful where the type classes (within a key class) are nearly equal to each other, because such relatively equal frequencies are most likely to exhibit clumping.

An alternative to this approach would be to process the entire file according to a default algorithm and then step the entire file through a CRT terminal where an analyst would make a judgment as to the appropriateness of the default algorithm to the individual members of the file. If the analyst decided that the applied algorithm were wrong, he would do a reverse purge (restore the data to its original form) and apply the default algorithm to the data, all with the press of a button. Data which were still incorrect could be corrected manually.

9.6 Pre-scanning and Mapping

The aim of the pre-scan should be to obtain from the data string readily extractable information which is highly relevant to the tasks to be performed by the algorithms. It should be very fast, as all data must undergo pre-scanning, and should "map" the string in order to obviate further re-scanning as much as possible. The map of a string will be a table which stores the locations (in the string) of the key characters (and keywords, etc.) encountered during the pre-scan, together with said key characters (and keywords, etc.).

Such maps will enable the translation of algorithmic sentences into object code. The algorithmic sentence compiler will have the job of properly linking to the maps the algorithmic operators which were derived from the sentences.

Experimentation will be required to determine what the optimal capabilities of the pre-scan should be. Although it is easy to write a pre-scan which detects only individual characters, the detection of strings of characters (dates, keywords) becomes a little bit more complicated and slower.

Should the pre-scan also be able to detect discontiguous patterns as single conditions? (E.g., a comma followed by a date to be considered as a unit; a semi-colon not followed by a word "edited" to be considered as a unit.) Such capabilities require more programming effort, slow down the pre-scan, and permit the pre-scan to take on some of the characteristics of the complex algorithms we are trying to replace. However, it may be decided that the utility of such capabilities may outweigh their disadvantages.

9.7 Automatic Language Recognition

Though it was not possible to simulate the performance of the AFRAD system in any exact sense, the quantitative approach was used in the development of algorithms for the recognition of language. First, some statistics were gathered from a test file. Then, algorithms based upon these statistics were developed, tested, and refined. The following general data was obtained:

a. Of the first 551 titles in the test file, 59% were English titles.

b. In a sample of the first 228 foreign language titles of the file, 78% had special symbols or diacritics in title and place, publisher, date string.

c. Based on the first 200 English titles of the file, the most frequent 1, 2, and 3 letter words (in decreasing frequency) are: *the, and, in, a, of, by, an, to, for.*

d. "A" is also found in Spanish, Portuguese, and Italian.

e. "In" and "an" also are found in German.

Figure 81 gives short words (1-3 characters) commonly found in several foreign languages.

9.7.1 Subroutine #1

This subroutine, scanning a given string of data until it hits a diacritic or special character, is very short and very fast. If the diacritic or special character is unique to a particular language, the code for that language is assigned to the string. If the diacritic or special character is found in two or more languages, the code for that language (of the given set) which is most frequent in the file is assigned as the default code. If no diacritic or special character is encountered in the string, the code for English is assigned as the default code. This subroutine has not yet been timed.

Based on a sample of the first 1000 records on the AFR MARC tape, this algorithm has an overall accuracy of about 81%. 98% of the English titles were identified as such. 72% of all foreign titles were recognized as being foreign language, and 15% of the foreign titles were mistakenly classified as English titles.

Of the titles, 619 are English, 381 are foreign. Of the 381 foreign titles, 220 were identified correctly as to actual language. Foreign languages correctly identified were: German, French, Spanish, Portuguese, and Czech. Potentially, with a few minor modifications to the tables in some cases, the following languages are recognizable: Danish, Swedish, Latvian, Polish, Hungarian, Turkish, and Indonesian.

9.7.2 Subroutine #2

This subroutine is very complicated and much slower relative to subroutine #1. The flow chart is given in Figure 82. The heart of the program is a table called DIATABLE, containing 86 fullwords (see Figure 80). Each fullword corresponds to a special character. Bit positions 2-24 correspond to 23 languages which may occur in the file. For example, German is assigned bit position 2, French is assigned bit position 3, and so forth all the way down to Albanian, which is assigned bit position 24. These languages are arranged in order of their presumed frequency in the AFR MARC File, with the more frequent languages being ahead of the less frequent ones. This order is important because it determines which language will be the default language in a given situation.

For the fullword in DIATABLE corresponding to a given diacritic-alphabetic combination or a given special character, the corresponding bit position is turned on for each language in which the given diacritic-alphabetic combination or special character may occur. For example, the eleventh word of DIATABLE is for the Diacritic-alphabetic combination A-GRAVE. This combination is found in the following languages: French, Italian, Portuguese, Catalan, Tagalog, and Latvian. The bit positions corresponding to these languages are turned on (i.e., made 1). All other bits are turned off. This results in a fullword whose hexadecimal representation is: 2C007C00.

Bit position 1 is turned on only when the particular diacritic-alphabetic combination or special character is unique to one language. Some of the words in DIATABLE are dummies, having the 1st and 31st bits turned on. Special characters which are usually found in non-Roman alphabets are routed to the 86th word, whose 1st and 32nd bits are turned on. Bit positions 25-30 are not used currently. Six additional languages, therefore, can be implemented in these positions.

The program operates in the following manner:

Step 1: A left-to-right scan is made of the data string. If a special character or diacritic-alphabetic combination is encountered, it is tested to see that it is legitimate. Illegitimate characters or combinations will generate a BAD DATA message, and the scan terminates.

Step 2: If legitimate, the corresponding fullword from DIATABLE will be "anded" with register 8, which initially contains all ones.

Step 3: The first bit of the fullword from DIATABLE is then tested to see if it is a one. If so, that special character or diacritic-alphabetic combination is unique to one language. The string is therefore scanned for English keywords. If one is found, ENG is assigned as the code. If not, the code for the unique language is assigned to the string.

Step 4: If the bit tested at step 3 was zero, the scan continues for additional diacritics and special characters. Proceed as in steps (1)-(3) until the end of string is encountered or one of the exception conditions above occurs.

Step 5: At the end of a complete scan of the string, if any diacritics or special characters have been encountered, then R8 will contain a one bit for each language in which the particular combination found in this string could possibly occur.

Step 6: Register 8 is then tested to see if it is zero. If so, then the string contains contradictory discritics or special characters. A message to this effect is generated, and processing terminates. This could occur, for example, if a string somehow contained an acute accent and an angstrom. There is no language which uses both the acute and the angstrom. As a result, the fullwords corresponding to the acute and angstrom, when "anded" into R8, would produce a zero result in R8.

Step 7: If, after testing, R8 is found not to be zero, the routine proceeds to link together a corresponding language block for each one bit in R8. In this manner, from one to 23 language blocks may be linked. Each language block contains up to 4 each of 1-, 2-, and 3-letter words commonly used in the particular language and usually unique to it. Please note that no words at all have yet been inserted in some of the language blocks (for example, Rumanian). This needs to be done. The language of the first block linked is assigned as the default code.

Step 8: The input string is now scanned for 1-, 2-, and 3-letter words, which are compared against the words in the linked blocks. If a match occurs, the language of the block in which the match occurred is taken as the language of the string. If no match occurs, the default code is assigned.

Step 9: If the scan of the string in step 1 encountered no diacritics or special characters, then the string is scanned for English key words. If one is found, the string is considered to be English. If none are found, the fullword located at FIVELANG is loaded into R8. This fullword can be set to any combination of ones and zeros. Control then passes to step 7, where all of the language blocks corresponding to the one bits in FIVELANG will be linked, and the string again scanned for 1-, 2-, and 3-letter words as described above.

This routine has not been timed yet. Hopefully, it is apparent to the reader that several modifications can be made to the algorithm to increase or decrease its accurary and speed. In general, greater accuracy means lower speed, although nothing is known about the quantitative relation between the two.

FIG. 80: UNIQUE DIATABLE DISPLACEMENT VALUES
(for the current 23 Roman alphabet foreign languages used in Subroutine 2)

	a	e	i	ø	u	c	g	n	s	r	y	l	z	t	d	l	g
ACUTE	0	1	2	3	4	7			6		9		8				
GRAVE	0	1	2	3	4												
DIERESIS	0	1	2	3	4												
BREVE	0				4		1										
CIRCUM-FLEX	0	1	2	3	4												
CIRCLE	0				4												
CEDILLA						7			6			0					
LEFT HOOK								5	6	3				2		0	1
RIGHT HOOK	0	1	2	3	4												
HECEK		1				7		5	6	3			8	2	1		
TILDE	0	1	2	3	4			5									
SUPERIOR DOT		1											8				

FIG. 81: SHORT WORDS COMMONLY FOUND IN SEVERAL FOREIGN LANGUAGES

Language	3-Letter Words	2-Letter Words	1-Letter Words
GER	die, das, der, von, und: den, dem, ein	fu ("r)	
FRE	les, par, une, des	du, le, et: un, de, la	
SPA	las, los: uno, una, por	al, el: un, la, de, en	y:a
ITA	dal, gli, per, un'	da, di, il, lo: in, le, l'	i: a, e, o
POR	aos, uma: por	as, do, os, um: em, ao, da	:a, e, o
DUT	een, met, van		
DAN	: ein, den, det, til	af, et, eu, og: de	
HUN	és, egy	az	
SWE	ett, med, och: den, det	en, av: de	i
PLS			i,w: k
CHE		:na	v: a, k
CRØ	nad, pod, prd, rad	do, sa, za:od, po	o, s, u:k
SLV		: od, po, za	:o, k
TAG	ang, kay	at, ni	
ICE	hin, hit: ein, til	ag:af	:i
FIN		ja	

Notes:

a. Words preceding colon are "keywords" for the given language.

b. Words following a colon are not unique to the given language. They may sometimes be used as keywords in another language of the same family which is considered to be more frequent in the file, but which is not the default (most frequent) language of the family. For example, Danish is the most frequent of the Scandinavian languages. Yet the word "i" is used as a keyword for Swedish because Swedish is considered to be much more frequent than the other Scandinavian language which uses this word, which is Icelandic.

c. Keywords have not yet been obtained for the following languages: Indonesian, Rumanian, Turkish, Catalan, Latvian, Afrikaans, and Albanian.

d. Additional keywords are needed for most of the languages listed in the table. The second version of the language recognition subroutine is designed to contain up to four 3-letter words, four 2-letter words, and four 1-letter words for each language.

FIG. 82:
LANGUAGE RECOGNITION ALGORITHM #2 FLOW CHART

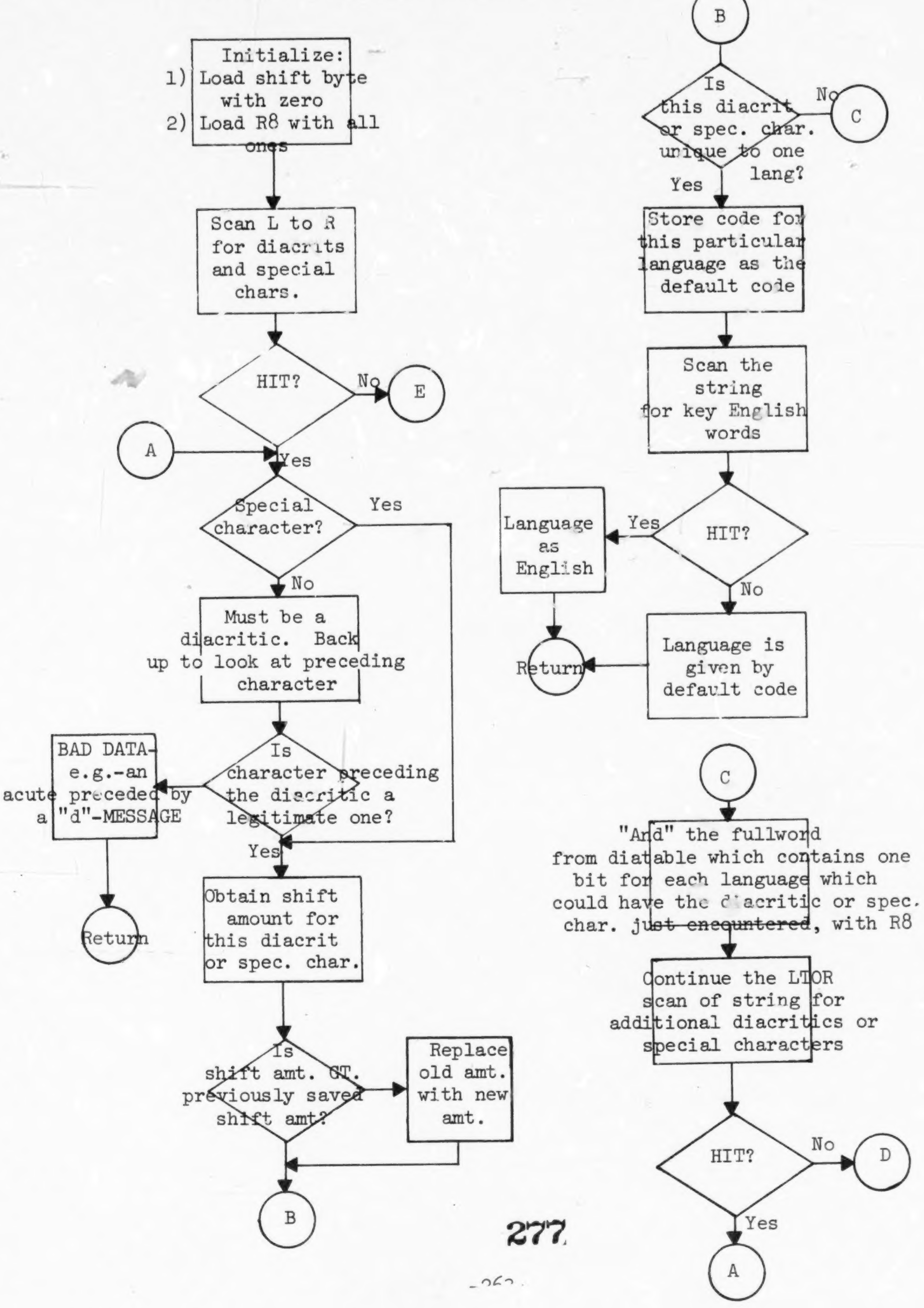
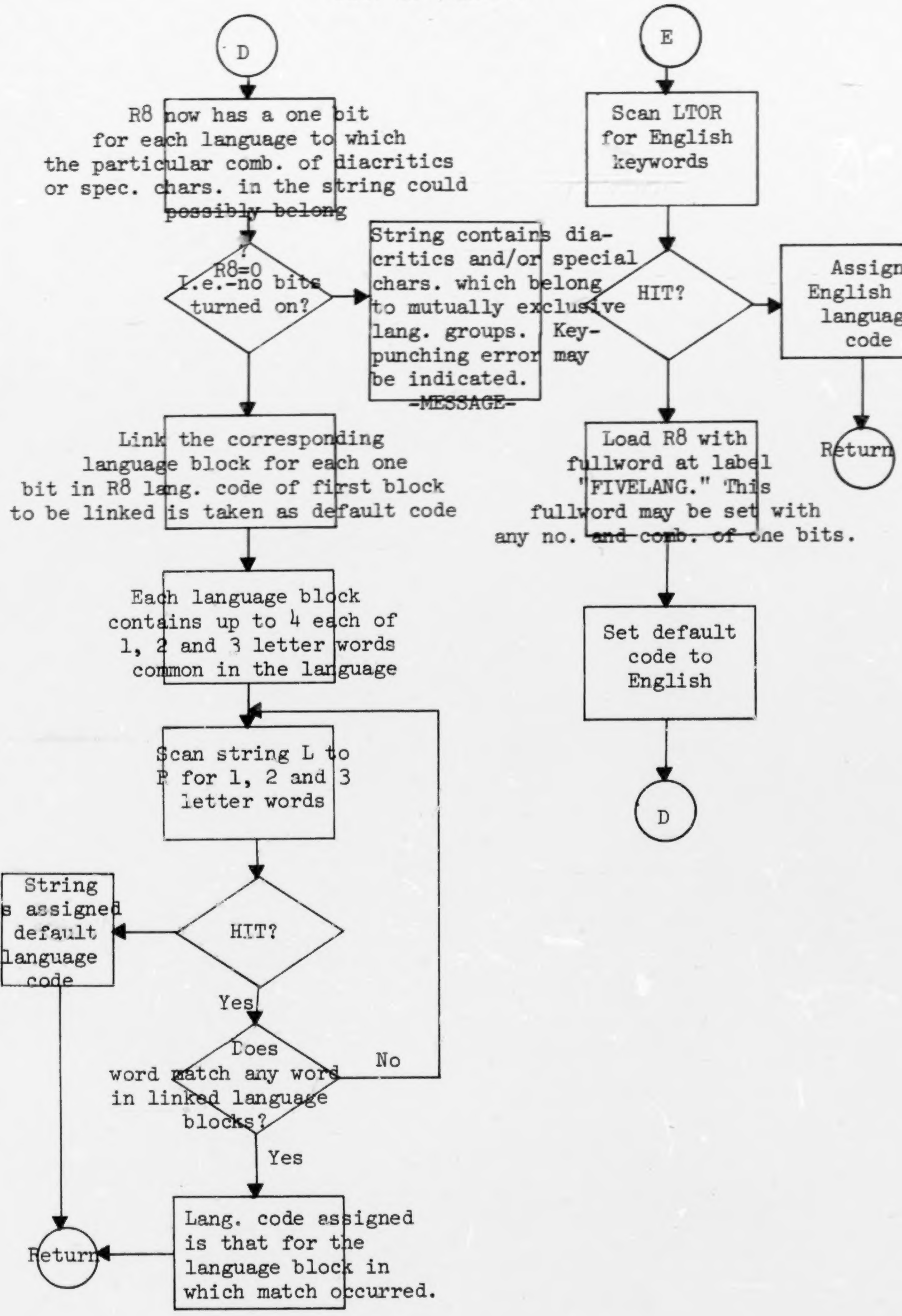

Initialize:
1) Load shift byte with zero
2) Load R8 with all ones
Scan L to R for diacrits and special chars.
HIT?
No
E
A
Yes
Special character?
Yes
No
Must be a diacritic. Back up to look at preceding character
Is character preceding the diacritic a legitimate one?
BAD DATA- e.g.-an acute preceded by a "d"-MESSAGE
Return
Yes
Obtain shift amount for this diacrit or spec. char.
Is shift amt. GT. previously saved shift amt?
Replace old amt. with new amt.
B
B
Is this diacrit or spec. char. unique to one lang?
No
C
Yes
Store code for this particular language as the default code
Scan the string for key English words
HIT?
Yes
Language as English
Return
No
Language is given by default code
C
"And" the fullword from diatable which contains one bit for each language which could have the diacritic or spec. char. just encountered, with R8
Continue the LTOR scan of string for additional diacritics or special characters
HIT?
No
D
Yes
A

FIG. 82 (Cont.):


D
R8 now has a one bit for each language to which the particular comb. of diacritics or spec. chars. in the string could ~~possibly belong~~
? R8=0 I.e.-no bits turned on?
String contains diacritics and/or special chars. which belong to mutually exclusive lang. groups. Keypunching error may be indicated.
-MESSAGE-
Link the corresponding language block for each one bit in R8 lang. code of first block to be linked is taken as default code
Each language block contains up to 4 each of 1, 2 and 3 letter words common in the language
Scan string L to R for 1, 2 and 3 letter words
HIT?
String is assigned default language code
Yes
Does word match any word in linked language blocks?
No
Yes
Lang. code assigned is that for the language block in which match occurred.
Return
E
Scan LTOR for English keywords
HIT?
Assign English as language code
Return
Load R8 with fullword at label "FIVELANG." This fullword may be set with any no. ~~and comb. of~~ one bits.
Set default code to English
D

10. TRANSLATION FROM ILR PROCESSING FORMAT TO MARC II COMMUNICATIONS FORMAT

by Jay L. Cunningham

10.1 Introduction

10.1.1 Purpose

A program, named LURCH, was designed to convert files of bibliographic data records from an obsolete version of the MARC II format to a very close approximation of the latest, full MARC II format. The obsolete records were created by the ILR program named "INFOCAL." The files created by INFOCAL will serve as input to LURCH. The output of LURCH will be files of records, one-for-one, in the UC MARC format.

10.1.2 Scope

LURCH, as presently designed, is intended for use with two particular files: the San Diego Biomedical Library book catalog source file, and the Santa Cruz holdings records file. However, any file output from INFOCAL can be processed by the program.

10.1.3 Contents

This description proceeds from a general narrative overview of the program, to a set of intermediate level flow charts, to a detailed definition of the modifications to be accomplished.

10.2 Narrative Description of Overall Program Structure

10.2.1 Housekeeping

Initialize buffers and tables for input record area, output record area, for old and new record directories, and for receiving new indicator values. Set up Tag Translation. (See Fig. 83 for table content and structure.)

10.2.2 Read the next INFOCAL Record to an Input Buffer

10.2.3 Shredout of INFOCAL Record Directory

First extract current record's length, base address, source agency (Santa Cruz or San Diego), and number of directory entries, for use in subsequent routines.

10.2.3.1 Major Loop

Extract each tag after the first, which will always be "000" (fixed fields). In sequence, tag by tag, and using the tag as the index to the Tag Translation Table, do a look-up to determine the identity and disposition of each field (i.e. keep same tag number, or replace with new tag number, or delete the tag, or delete both tag and field). Also,

for each tag, find new indicator values, using old tag and old Indicator 1 (in INFOCAL directory) as arguments. Extract old field length and old starting character position from directory entry. Compute right-end address for later use in moving the fields. The old tag and old Indicator 1 are posted incidentally to a table but this is not strictly necessary, once their look-up functions have been performed. See Fig. 84 for the format of this table (used in the Fortran version of the program).
In the same major loop, a call or a branch is next made to a subroutine or program segment, depending upon the outcome of the look-up of the current tag in the Tag Translation Table. For each tag in the table, a flag or signal controls the program flow depending upon the tag's identity. For example, in the simplest case (default), nothing is done; the old tag number is copied to the new directory table, preparatory to copying it to the new directory area in the output buffer. At this time there are ten of these segments. At the end of the logic for each tag, the new field length and new starting character position are computed, the new tag number is copied to the new directory table, and the new indicator values likewise are copied to a table preparatory to being copied to the proper variable fields in the output buffer. This ends this major loop, when all tags in the current input record's directory have been processed. (At this point no data have been moved.)

10.2.3.2 Bookkeeping

Prior to moving anything, the new directory table must be systematically adjusted to reflect the outcomes of the changes found to be needed in the previous loop. For example, the field length of data items which were previously tag fields but are now subfields, must be added to the length of the new "parent" field.

10.2.4 Major Loop

Move variable fields data from input buffer to output buffer. An output buffer pointer is computed as a result of knowing now the length of the new directory, which was computed in the previous major loop. Depending upon the file (San Diego or Santa Cruz) and the tag number, the loop proceeds through the new directory table, tag by tag, copying the new values for Indicators 1 and 2 into the positions formerly occupied by INFOCAL Indicators 6 and 7, inserts the $a, transfers the variable field data for the new field, and inserts the field terminator. The last field terminator in the current record is replaced by a record terminator.

10.2.5 Create Fixed Field 008

This field is comprised of elements extracted and slightly rearranged from the 000 Fixed Field in the input record.

10.2.6 Create New Leader

This field is comprised of elements extracted from the leader and 000 Fixed Fields in the input record.

10.2.7 Create New Directory

The new directory entries are copied to this area of the output buffer from the new directory table.

10.2.8 Write the Completed Record to Output Storage

Return to read the next record (10.2.2).

10.2.9 End of File Check

Stop.

FIG. 83:
TAG TRANSLATION TABLE

Position (Implied) (3)*	New Tag Number (3)	Action Code for Special Processing (2)	Pattern # Indicator Key (2)
000	008	b̸	b̸
.	.	.	.
.	.	.	.
003	041	01	01
.	.	.	.
.	.	.	.
.	.	.	.
090	b̸	02	b̸
.			
.			
.			
100	b̸	03	02
.			
.			
.			
.			
.			
899			

*Number of character positions in column of table.

FIG. 84:
DIRECTORY RE-BUILDING CONTROL TABLE

Shredout of Old Record Directory — Creation of New Directory and Indicators

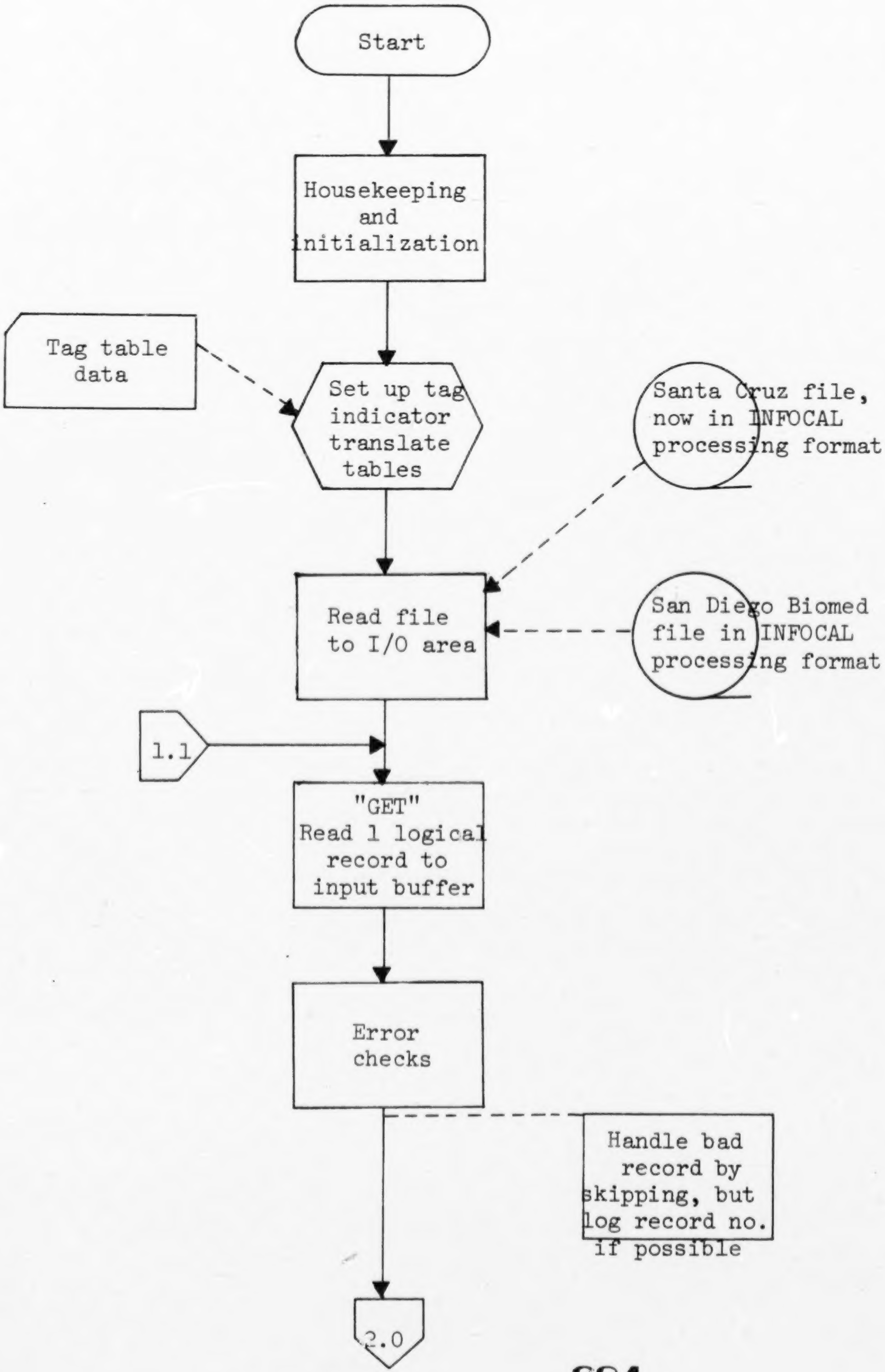
Old Directory and Indicators					New Directory and Indicators				
Tag	Ind. 1	Field Length	Starting Character Position	Right Address	Tag	Field Length	SCP	Ind. 1	Ind. 2
000	b̸	26	00	25	007	0026	0000	b̸	b̸
003	0	05	26	30	008	41	26	b̸	b̸
090	b̸	28	31	58	Ø*	b̸	b̸	b̸	b̸
100	1	18	59	76	090	24	67	"j"	b̸
240	0	90	77	166	100	21	91	1	0
.					245	93	112	1	b̸
.					.				
.					.				
660	b̸	17	281	297	.				
664	b̸	09	298	306	660	29	331	b̸	b̸
.					999*	[09]**	0	b̸	b̸
.					.				
.					.				

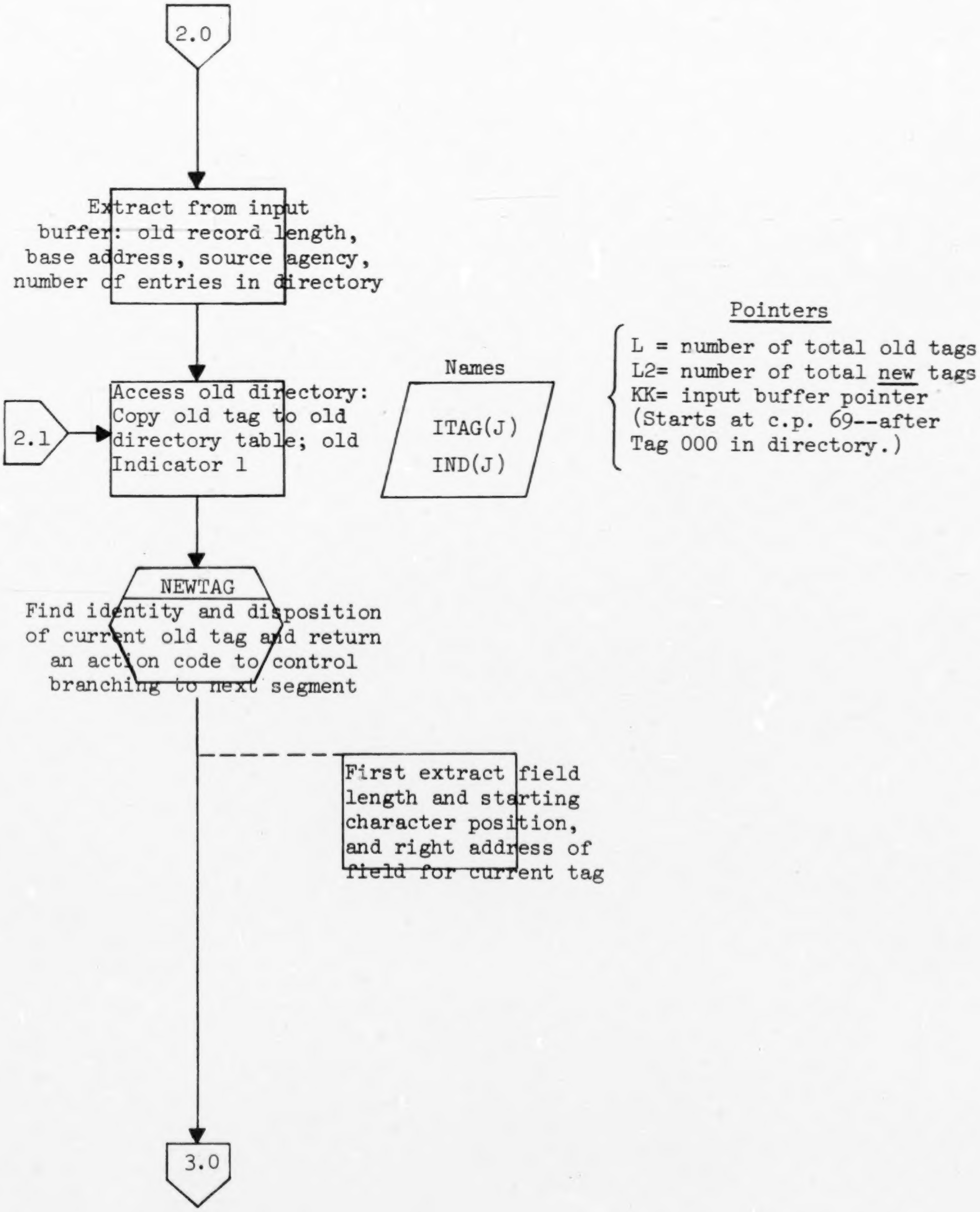
*"Ø" means "delete both the tag and the variable field data" (currently applies only to language code). "999" means "delete tag only"; data becomes a subfield under most recent preceding "parent" tag.

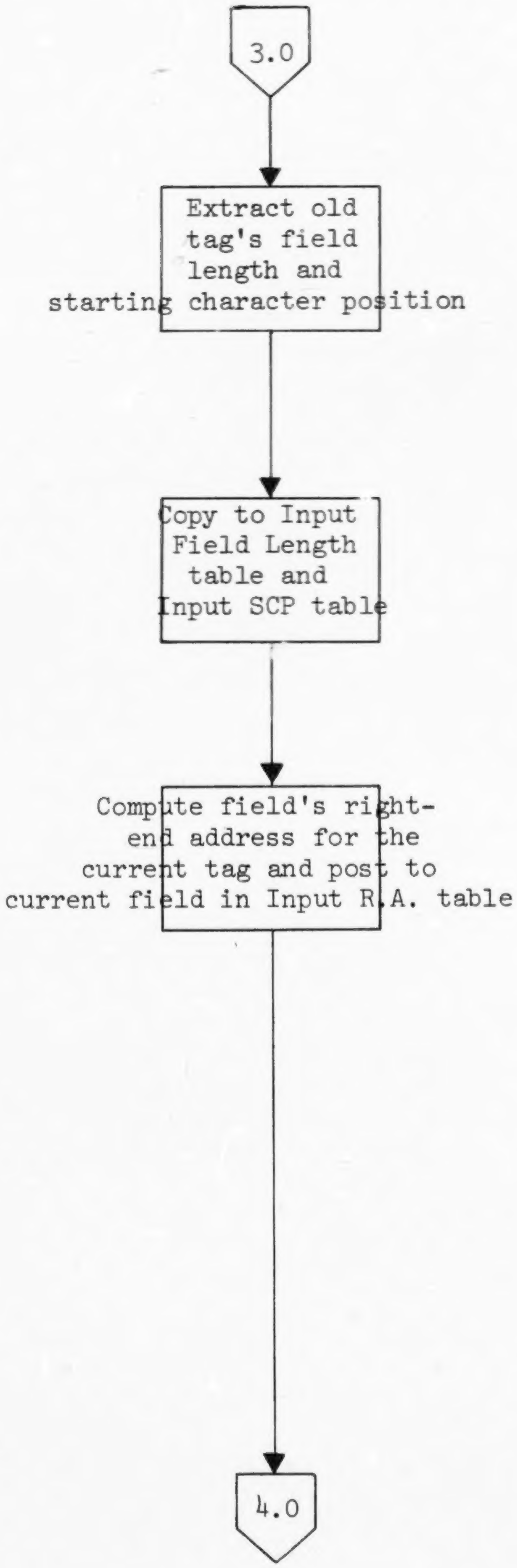
**FL value is added to preceding "parent" FL.

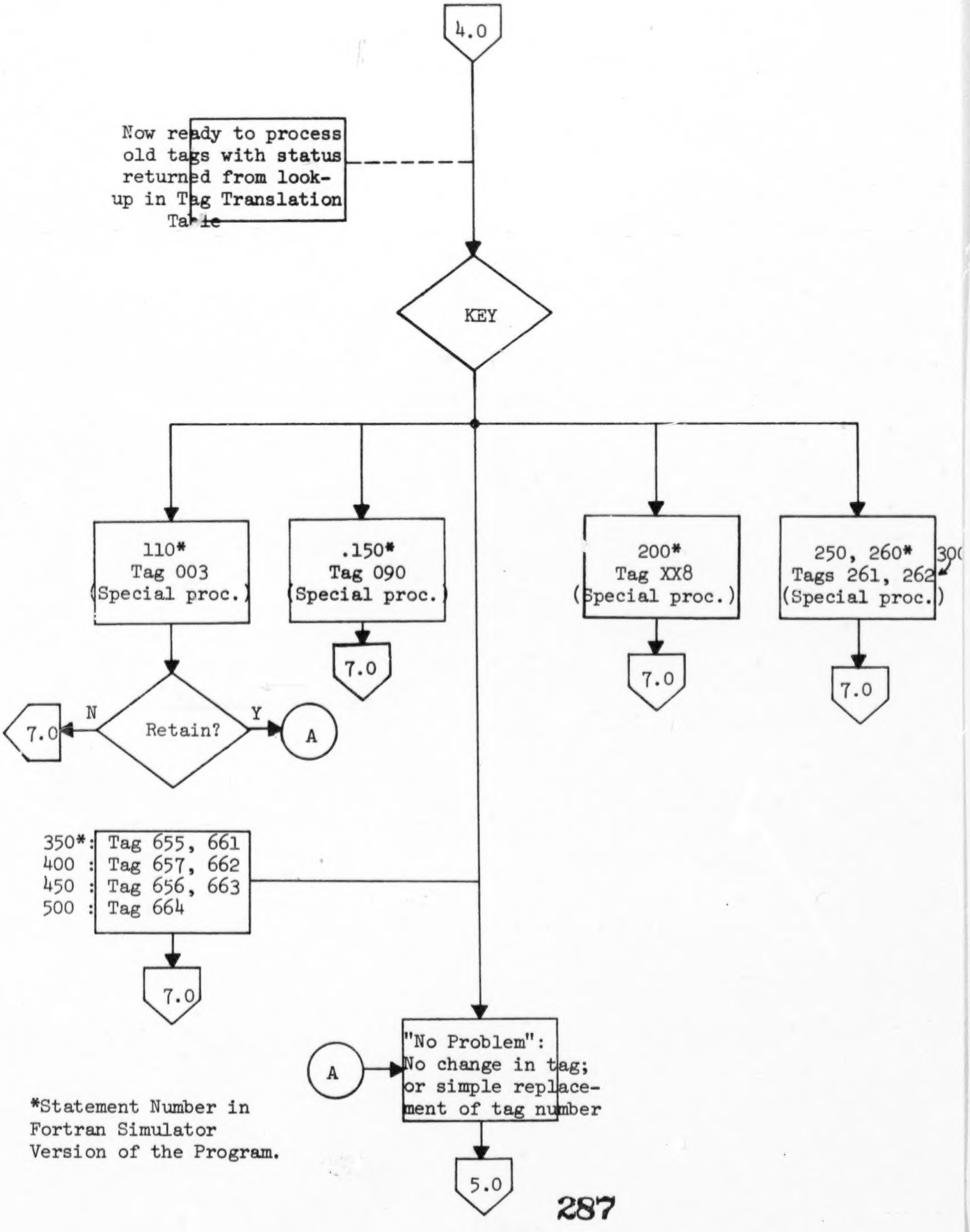
10.3 Flowcharts

PROGRAM "LURCH"

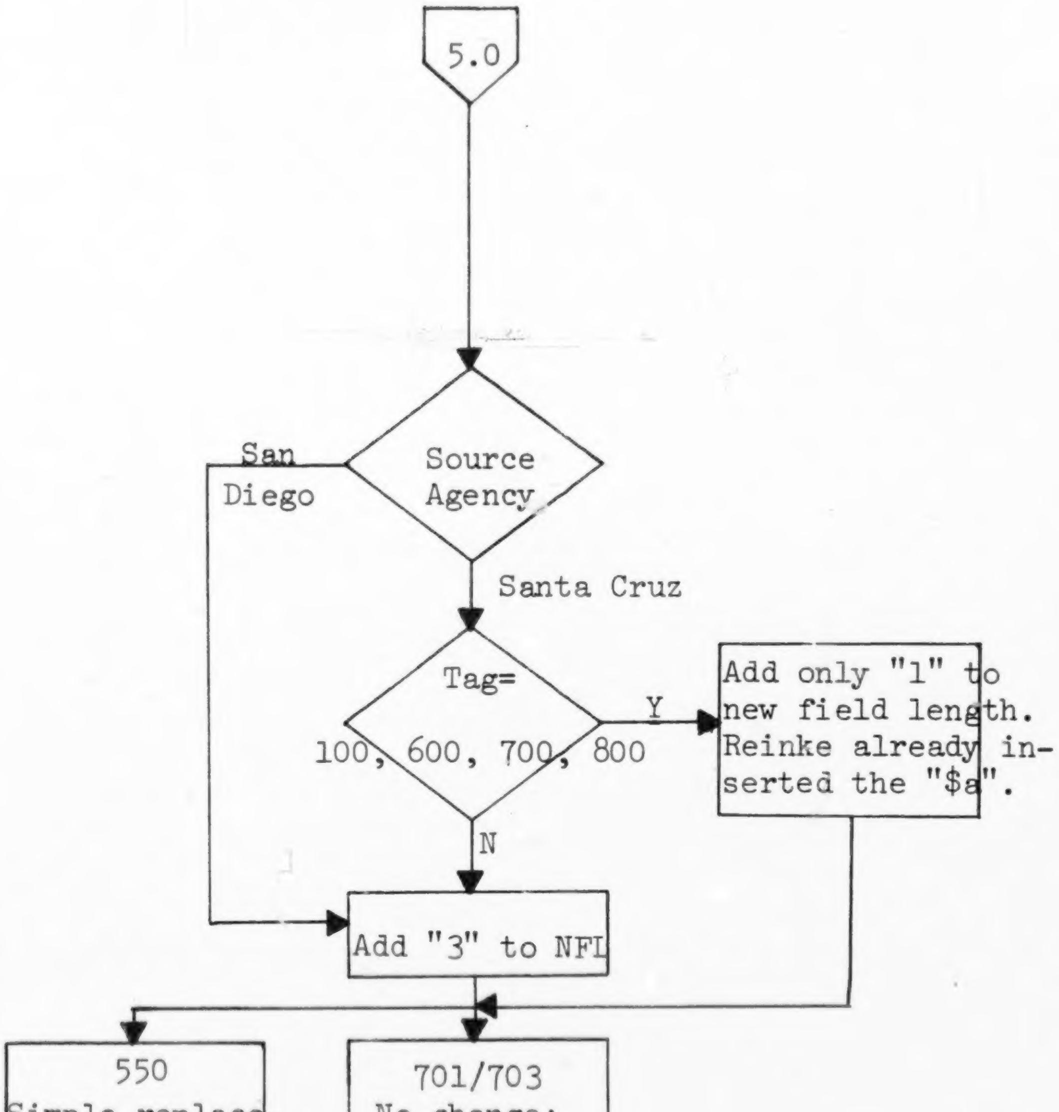

Start
Housekeeping and initialization
Tag table data
Set up tag indicator translate tables
Santa Cruz file, now in INFOCAL processing format
Read file to I/O area
San Diego Biomed file in INFOCAL processing format
1.1
"GET" Read 1 logical record to input buffer
Error checks
Handle bad record by skipping, but log record no. if possible
2.0

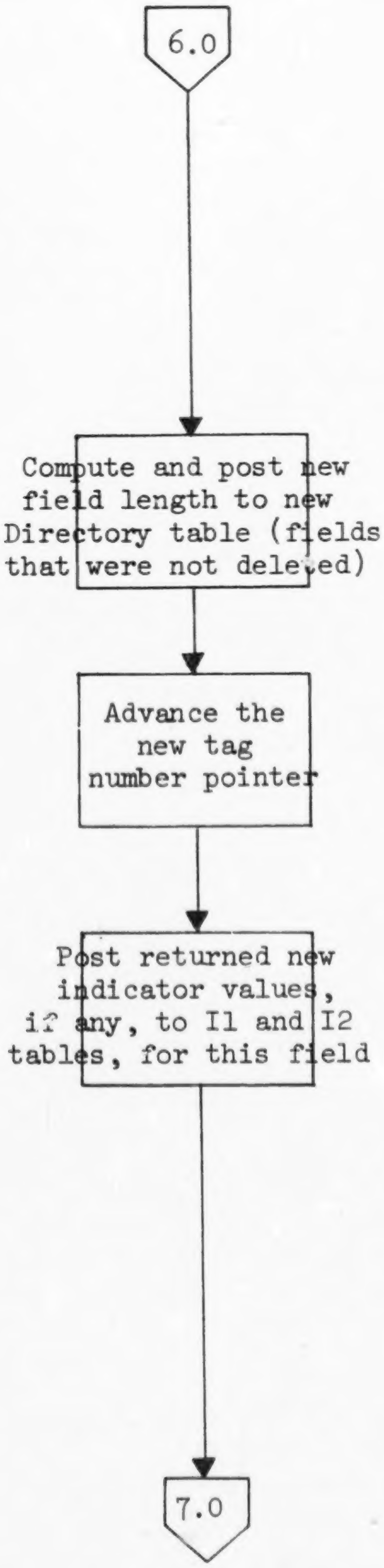

2.0
Extract from input buffer: old record length, base address, source agency, number of entries in directory
2.1
Access old directory: Copy old tag to old directory table; old Indicator 1
Names
ITAG(J)
IND(J)
Pointers
L = number of total old tags
L2= number of total new tags
KK= input buffer pointer (Starts at c.p. 69--after Tag 000 in directory.)
NEWTAG
Find identity and disposition of current old tag and return an action code to control branching to next segment
First extract field length and starting character position, and right address of field for current tag
3.0

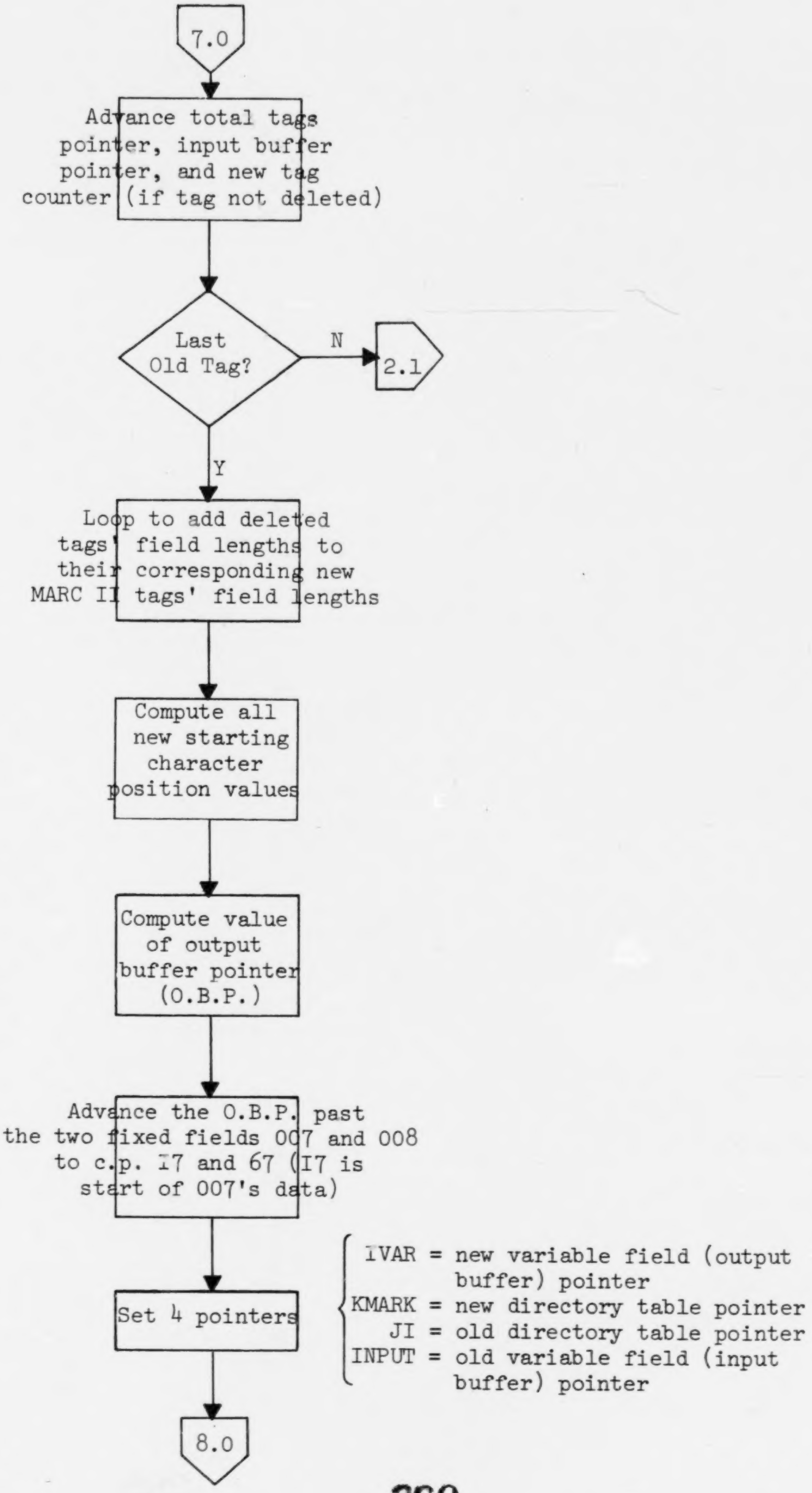

3.0
Extract old tag's field length and starting character position
Copy to Input Field Length table and Input SCP table
Compute field's right-end address for the current tag and post to current field in Input R.A. table
4.0


4.0
Now ready to process old tags with status returned from look-up in Tag Translation Table
KEY
110*
Tag 003
(Special proc.)
.150*
Tag 090
(Special proc.)
200*
Tag XX8
(Special proc.)
250, 260*
Tags 261, 262
(Special proc.)
30
7.0
7.0
7.0
7.0
N
Retain?
Y
7.0
A
350*: Tag 655, 661
400 : Tag 657, 662
450 : Tag 656, 663
500 : Tag 664
7.0
A
"No Problem":
No change in tag;
or simple replacement of tag number
5.0

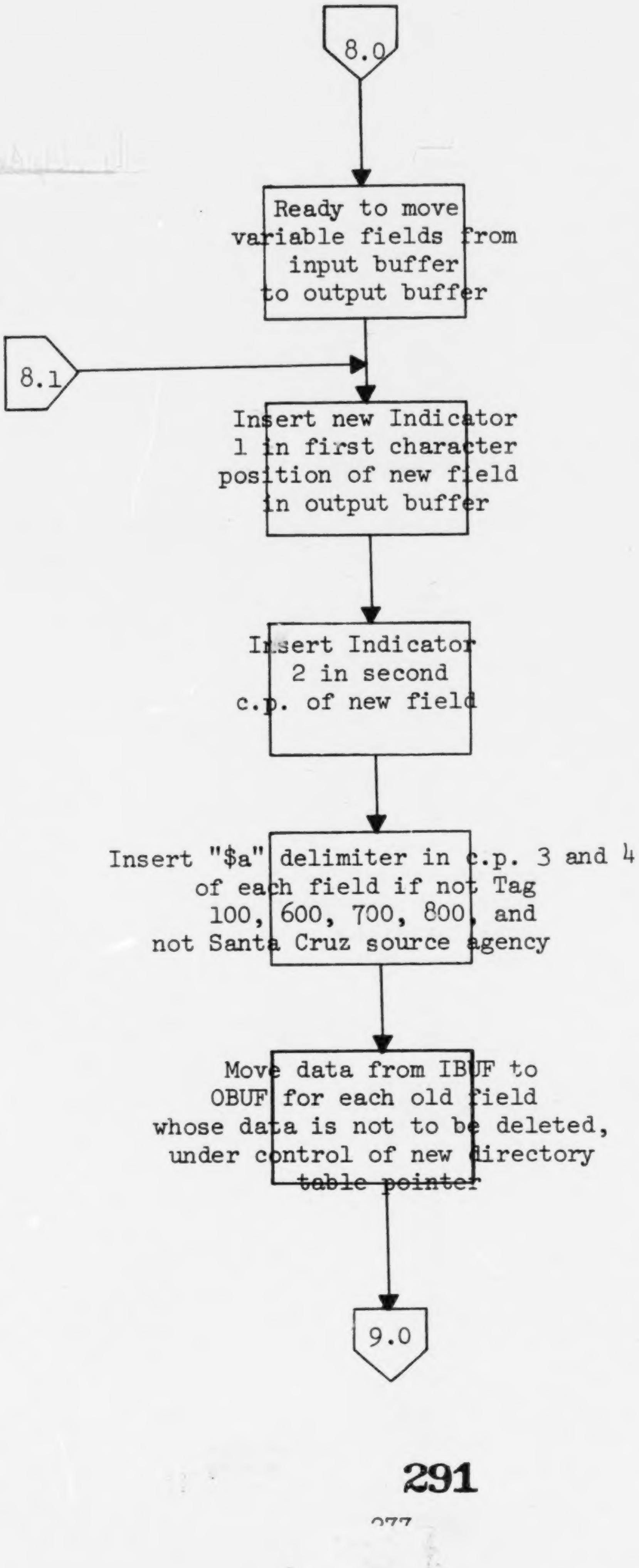
*Statement Number in Fortran Simulator Version of the Program.

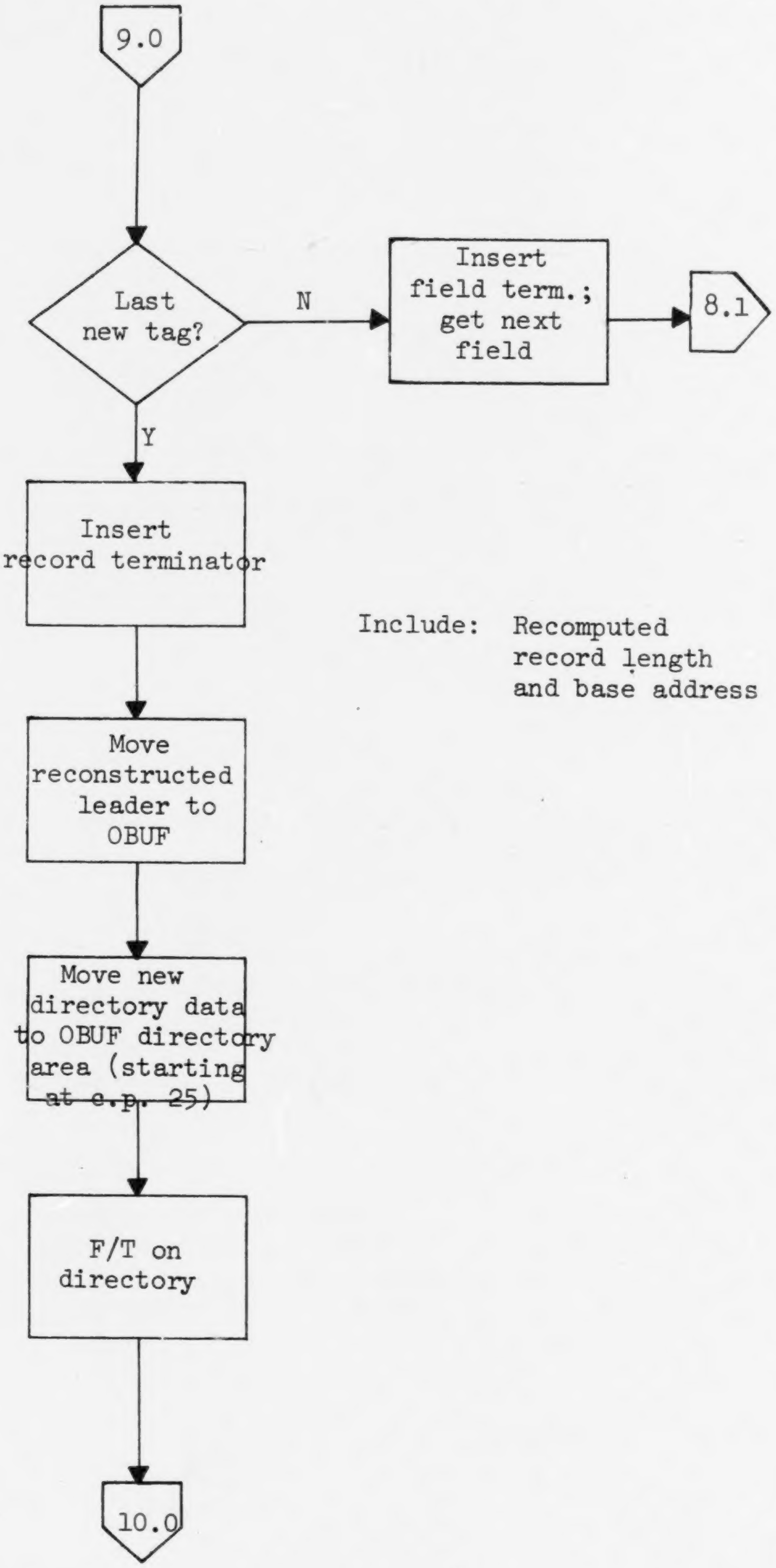

5.0
Source Agency
San Diego
Santa Cruz
Tag= 100, 600, 700, 800
Y
N
Add only "1" to new field length. Reinke already inserted the "$a".
Add "3" to NFL
550 Simple replacement: Copy new tag to new Dir. table
701/703 No change: copy old tag to new Dir. table
6.0

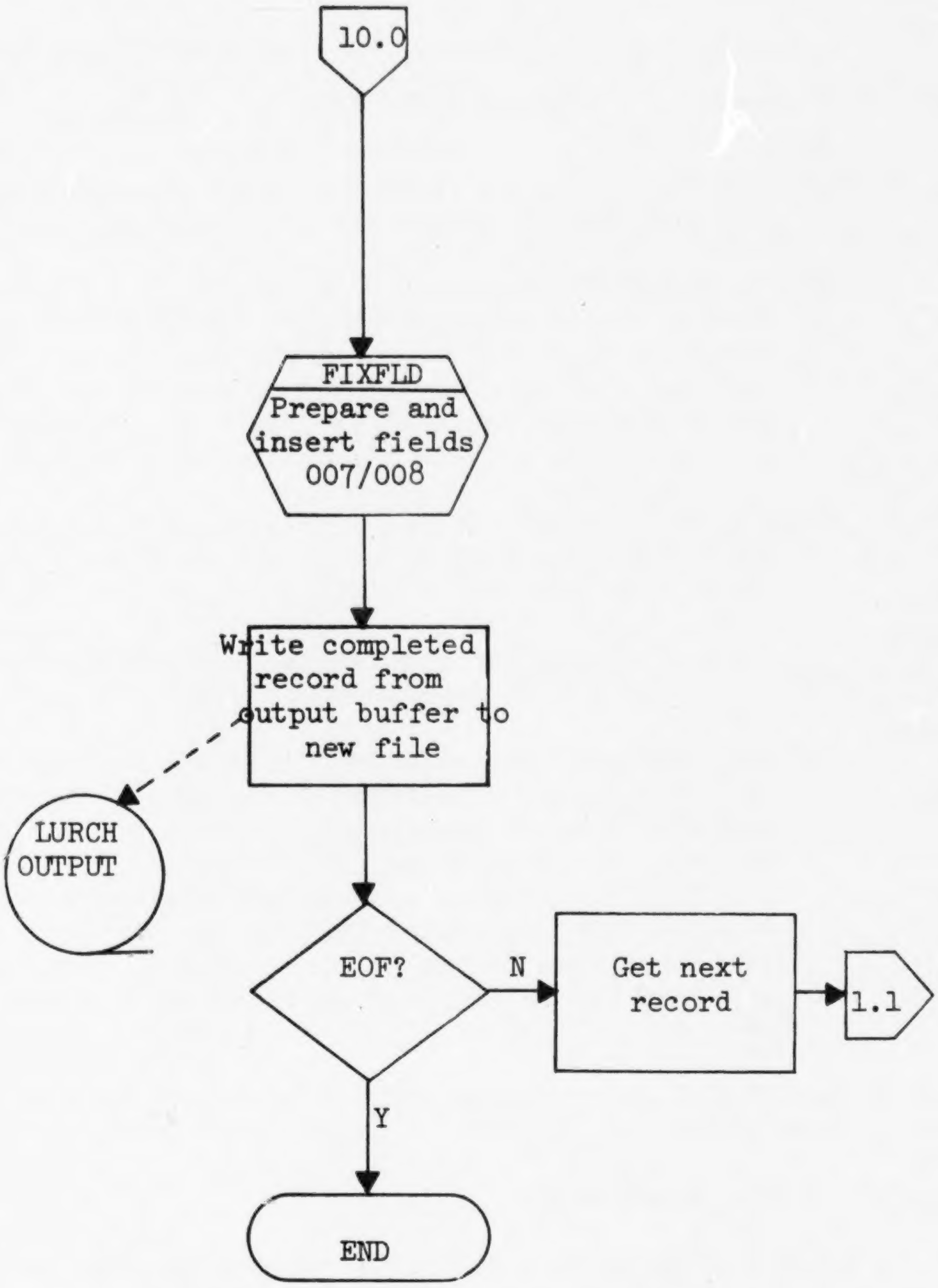

6.0
Compute and post new field length to new Directory table (fields that were not deleted)
Advance the new tag number pointer
Post returned new indicator values, if any, to I1 and I2 tables, for this field
7.0


7.0
Advance total tags pointer, input buffer pointer, and new tag counter (if tag not deleted)
Last Old Tag?
N
2.1
Y
Loop to add deleted tags' field lengths to their corresponding new MARC II tags' field lengths
Compute all new starting character position values
Compute value of output buffer pointer (O.B.P.)
Advance the O.B.P. past the two fixed fields 007 and 008 to c.p. I7 and 67 (I7 is start of 007's data)
Set 4 pointers
8.0

IVAR = new variable field (output buffer) pointer
KMARK = new directory table pointer
JI = old directory table pointer
INPUT = old variable field (input buffer) pointer


8.0
Ready to move variable fields from input buffer to output buffer
8.1
Insert new Indicator 1 in first character position of new field in output buffer
Insert Indicator 2 in second c.p. of new field
Insert "$a" delimiter in c.p. 3 and 4 of each field if not Tag 100, 600, 700, 800, and not Santa Cruz source agency
Move data from IBUF to OBUF for each old field whose data is not to be deleted, under control of new directory table pointer
9.0


9.0
Last new tag?
N
Insert field term.; get next field
8.1
Y
Insert record terminator
Include: Recomputed record length and base address
Move reconstructed leader to OBUF
Move new directory data to OBUF directory area (starting at c.p. 25)
F/T on directory
10.0


10.0
FIXFLD
Prepare and insert fields 007/008
Write completed record from output buffer to new file
LURCH OUTPUT
EOF?
N
Get next record
1.1
Y
END

10.4 Summary of MARC II Record Structure (Communication Format)

1.	2.	3.	4.
Leader	Record Directory	Control Fields	Variable Fields

1. LEADER--The leader is fixed in length for all records and contains 24 characters.

2. RECORD DIRECTORY--The record directory is made up of a series of fixed-length entries (twelve characters each) which contain the identification tag, the length, and the starting character position in the record of each of the variable fields. The record directory will end with a field terminator code ($1E_{16}$).

3. CONTROL FIELDS--The control fields contain alphameric data elements, many of which have a fixed length. These fields end with a field terminator code. Each control field is identified by a 3-character numeric tag in the record directory, and these tags will not be repeated in a logical record.

4. VARIABLE FIELDS--The variable fields are made up of variable length alphameric data, and all fields end with a field terminator code except the last variable field in a logical record which replaces the field terminator with an end-of-record code ($1D_{16}$). Each variable field is identified by a 3-character numeric tag in the record directory, and tags may be repeated as required in a logical record.

10.4.1 Modifications to Overall Record Structure to Produce a Revised MARC II Record

10.4.1.1 Field Terminators

To bring the Santa Cruz file into conformity with MARC II, after it has been run through the INFOCAL program, the following actions should be taken to update the structure of each record:

a. Insert a field terminator code ($1E_{16}$) at the end of the record directory.

b. Insert a field terminator code at the end of each of the control fields (Tag 008 in MARC II; and Tag 007 in addition, if used).

c. Insert a field terminator code at the end of each variable field, except the last variable field in a logical record which will contain an end-of-record code in place of the field terminator code. The end-of-record code is "$1D_{16}$."

10.4.1.2 Subfield Delimiter Codes

In some cases, subfield delimiter codes have been inserted in the proper positions by the TRANSCOF program. However, as a final cleanup of the translated Santa Cruz file to conform to the latest MARC II, the following actions should be taken:

a. Insert, by a test on presence of the code in c.p. 3-4 of each variable field except those listed below, the delimiter "$1F_{16}$a" in c.p. 3-4 of each variable field, if it is not already there.

Exceptions:

(1) Tag 007 and Tag 008, the fixed length data elements fields. These fields do not have indicators or subfield delimiters.

(2) Tag 091, the revised field for Santa Cruz shelf key data. This field will not use the subfield delimiters.

b. Comply with such other instructions as to subfield coding as below specified, including those in Section 10.5 governing the coding of the Tag 090 holdings field for UC MARC.

Fields in which delimiters other than the first subfield ("$1F_{16}$a") have been inserted by the TRANSCOF program are listed in the tables in Section 10.2.

No further action on subfields is necessary in this phase of the file translation. Further subfield identification will be deferred until AFR development has progressed far enough to enable recognition of presently non-identified elements in the SC file.

10.4.1.3 MARC II Indicators 1 and 2

It will not be necessary to place space in the record for Indicators 1 and 2. Two spaces are already created by INFOCAL at the head of each variable field for this purpose. The two spaces now contain zeros in the INFOCAL record. The appropriate values for the indicators that apply to any given field must be sought in the "old" Indicator 1 position in the INFOCAL directory entry, and the value found must be translated and moved to the appropriate position, either in Indicator 1 ("new") or Indicator 2 ("new") in the variable fields. The logic for this translation is contained in Fig. 88.

10.4.2 Modifications to Leader

10.4.2.1 Structure and Content of MARC II Leader

Outline of Leader

0 4	5	6	7	8 9	10	11	12 16	17 23
Record Length	Status	Type of Rec.	Biblio. Level	Blanks	Indicator Count	Subfield Code Count	Base Address of Data	Blanks

Element Number	Name of Leader Data Element	Number of Characters	Character Position in Record
1.	Logical Record Length	5	0-4
2.	Record Status	1	5
3.	Legend		
	a. Type of Record	1	6
	b. Bibliographic Level	1	7
	c. Blanks	2	8-9
4.	Indicator Count	1	10
5.	Subfield Code Count	1	11
6.	Base Address of Data	5	12-16
7.	Blanks	7	17-23

NOTE:

Base Address of Data: a number which is the starting character position of the first control field. That is, it is equal to the length of the leader and the record directory (including the record directory field terminator).* The starting character position for each field entered in the record is relative to the first character of the first control field (not the beginning of the record). The base address of data gives the base from which each field is addressed. The number is right justified with leading zeros.

---

*The record character position count starts at position zero. Therefore if there are 50 characters in the leader and directory, the address of the first position of the first control field (data for Tag 007 in UC MARC) will be "50".

10.4.2.2 Changes in INFOCAL Leader - Translation Table

FIG. 85:
FORMAT TRANSLATION TABLE LEADER SEGMENT

FIELD NAME	OLD POS.	ACTION	NEW POS.	CONTENTS, REMARKS, ETC.
Record Length	0-4	NC*	0-4	5 EBCDIC digits
Status Date	5-10	DELETE		
Record Status	11	CHANGE CONTENTS & POSITION	5	
Legend Extension	12	DELETE		
Record Type	13	CHANGE POSITION	6	a = language material (printed)
Bibliographic Level	14-16	CHANGE LENGTH & POSITION	7	m = monograph OR s = serial
(Blanks)	--	ADD	8-9	
Indicator Count	17	CHANGE CONTENTS & POSITION	10	Count = 2
Subfield Code Count	--	ADD	11	Count = 2
Base Address of Variable Fields	18-19	RECOMPUTE & CHANGE POSITION	12-16	5 EBCDIC digits
ILR Master Record Number	39-45	CHANGE POSITION TO HERE	17-23	See old c.p. 39-45 7 EBCDIC digits, right just./left zeros
Origin of Record	20-22	DELETE		
Processor Date	23-28	DELETE		
Processor of Record	29-31	DELETE		

*NC = No Change

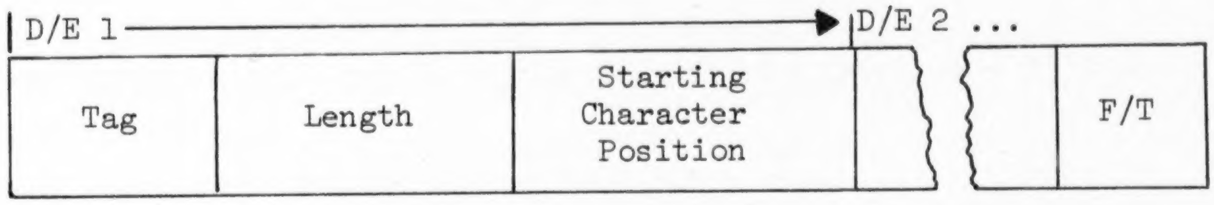
FIG. 85 (Cont.):
FORMAT TRANSLATION TABLE LEADER SEGMENT

FIELD NAME	OLD POS.	ACTION	NEW POS.	CONTENTS, REMARKS, ETC.
Source Type of Catalog Card	32	DELETE		
Agency of Source Type	33-35	DELETE		
Adapter of Catalog Card	36-38	DELETE		
Master Record Number (ILR-assigned)	39-45	CHANGE POSITION FROM HERE	17-23	7 EBCDIC digits in new c.p. 17-23
Check Sum on Record Number	46	DELETE		
Number Entries in Directory	47-49	DELETE		
Date Entered on Master File	50-55	MOVE TO FIXED LENGTH DATA ELEMENTS FIELD		See Fig. 89.

10.4.3 Modifications to Directory

10.4.3.1 Structure and Content of MARC II Directory

Outline of Record Directory Entries



F/T - Field Terminator ($1E_{16}$)

Element Number	Name of Record Directory Data Element	Number of Characters	Character Positions in Directory Entries
1.	Tag	3	0-2
2.	Field Length	4	3-6
3.	Starting Character Position	5	7-11

Contents of Record Directory Entries

1. Tag--a 3-character numeric symbol which identifies the field.

2. Field Length--the number of characters in the field identified by the tag. This count includes indicators, subfield codes, data, and field terminator. The number is right justified with leading zeros.

3. Starting Character Position--five numeric characters which give the character position in the record of the first character of the field. The character position is relative to a base which begins with the first character of the first field (i.e., for monographs the first character of the control number field). The first record directory entry will contain the starting character position. Subsequent record directory entries will have starting character positions incremented by the field length of the previous entry.

4. Mode--All data in a directory entry will be in EBCDIC.

Example:

Entry 1	001	0014	00000
Entry 2	008	0047	00014
Entry 3	100	0058	00061

It should be noted that in a variable field (as opposed to a control field) the first actual data character is the fifth character, i.e., the starting character position plus four. The starting character position number is right justified with leading zeros.

10.4.3.2 Changes in INFOCAL Directory

a. Structure of Directory. The record directory produced by INFOCAL will require extensive revision to bring it into consonance with LC MARC II.

In INFOCAL there were a total of seven indicator positions allocated. See Fig. 87. Five of these were placed in the directory entry. The remaining two were placed at the beginning of the variable field (filled with blanks if not applicable to any given field). The revision of the indicator structure is as follows:

FIG. 86:
INDICATOR REVISION

New Indicator No.	Old Indicator No.	Position	Disposition
None	1	Directory	Change contents according to Fig. 88 logic, and place transformed contents in new Indicators 1 and 2 at head of variable field, as needed.
"	2	Directory	DELETE.
"	3-5	Directory	DELETE.
1	6	Var. Field	Place contents resulting from transformation of old Indicator 1, according to logic in Fig. 88.
2	7	Var. Field	Same as 6.

The field length and starting character positions in the individual directory entry will have to be recomputed as a result of a change in the length of the fixed length data element field (the first variable field in every record), and the addition of field terminators and record terminator to every record. The formula for recomputation is given as part "c" of Section 10.4.3.2.

b. Content of INFOCAL Indicator 1 (In Directory). Each directory entry must be processed to change its structure. As part of this revision, the values resident in Indicator 1 (c.p. 3 in INFOCAL-produced

FIG. 87:
INFOCAL DIRECTORY ENTRY STRUCTURE

	Length	Char. Pos.	Content
Directory Entry 1	3	0	Tag (3 EBCDIC digits)
	1	3	Indicator 1 (EBCDIC character. If not used, blank.)
	1	4	Indicator 2 (Repeatable tag number, applicable to those tags which can appear more than once in a given record. If tag is not currently repeatable, indicator will be set to binary zero. An 8-bit binary digit.)
	1 1 1	5	Indicators 3, 4, 5 (Character positions provided for future expansion. Currently set to blanks.)
	2	8	Field Length (A 16-bit number giving the character length of the variable field, including Indicators 6 and 7.)
	2	10	Starting Character Position (A 16-bit number giving the position of the first character of the variable field. Currently the first character will always contain Indicator 6. This position is relative to the first character of the fixed length data elements field.)
	• • •		
Directory Entry n			
[End of Directory]			

The total length of the directory entry is 12 characters. The total length of the directory is 12 x the number of directory entries.

directory entries) must be moved to the variable field. The logic for this transformation is dependent on the tag number and the value detected. Only values found in the SC tapes are described here.

FIG. 88:
REORGANIZATION OF INDICATOR VALUES

TAG*	INFOCAL IND. 1 VALUE	DISPOSITION IN MARC II PLACE IN INDIC. 1	 PLACE IN INDIC. 2
003	0 = Single or Multi-Lang.	0 = Multi-Lang. (variable field present only if more than 1 lang. code)	b̸
	1 = Translation	1 = Translation	
100	1 = Single Surname, Not Subject	1	0
	5 = Single Surname Is Subject	1	1
110	2 = Name (Direct Order), Not Subject	2	0
	6 = Name (Direct Order), Is Subject	2	1
130	0 = Not Subject	b̸	0
200	1 = Will be Printed on LC Cards	1	b̸
210	1 = Make Title Added Entry	1	b̸
240	0 = No Title Added Entry	0	b̸
	1 = Make Title Added Entry	1	b̸
261	0 = Pub. Not Main Entry	0	b̸
410	2 = Name (Direct Order), Series Author Not Main Entry	2	0

---

*INFOCAL tag numbers are used in this table. For revised tag numbers to which these are to be changed, see Fig. 90.

FIG. 88 (Cont.)
REORGANIZATION OF INDICATOR VALUES

TAG	INFOCAL IND. 1 VALUE	DISPOSITION IN MARC II	
		PLACE IN INDIC. 1	PLACE IN INDIC. 2
490	0 = Series Not Traced	0	b̸
	1 = Series Traced Diff.	1	b̸
520	Not Used (set to b̸)	0*	b̸
600	1 = Single Surname	1	0**
650	Not Used (set to b̸)	0***	0**
700	a = Single Surname, Alternative	1	0
	e = Single Surname, Secondary	1	1
710	b = Name (Direct Order), Alternative	2	0
740	1 = Title Added Entry, Traced Diff.; Secondary	b̸	1
810	2 = Name (Direct Order)	2	b̸

---

*The new MARC II tag for contents note has values for Indicator 1 that show characteristics of the contents note. The "0" value, for "Contents" type of contents note will be used as the default value.

**The "0" value indicates Library of Congress Subject Headings List as the source of the subject heading in this field. This is a default setting.

***The new MARC II definition for the Tag 650 field has an Indicator 1 value for whether the heading starts with a place name. The value "0" meaning "not entered under place" will be used as the default setting.

c. Summary of Recomputations Required. The following computation must be undertaken to revise the value of the logical record length field in c.p. 0-4 of the leader, in each record output from the INFOCAL program run on Santa Cruz:

TRL = L + D + CF + VF

where

TRL = Total Record Length (Logical Record)
L = length of Leader
D = length of Directory
CF = length of Control Fields
VF = length of Variable Fields

As part of this recomputation of the value that will be present in the INFOCAL output record, the following fields must also be recomputed, either becuase of the insertion of new fields of data, or due to insertion of new codes such as field terminators, to achieve MARC II compatibility:

Base Address of Data (c.p. 12-16 in revised record leader)

Field Length of each control field and variable field (Note that these fields do not have indicators in c.p. 1 and 2)

Starting Character Position of each control and variable field

The following computation will yield the revised value for base address, when applied to each Santa Cruz record, after processing by INFOCAL:

Length of Leader = 24 characters

Length of Directory = 12*n, where n is the number of directory entries

Directory Field Terminator = 1

Base Address = 24 + (12*n) + 1

The following computations must be made to revise the field lengths and starting character position values of each entry in the record directory. Also convert from 16-bit binary to EBCDIC mode.

INFOCAL LENGTH COMPUTATION	REVISED LENGTH COMPUTATION
Indicator 6 = 1 position	Indicator 1 = 1 position

Indicator 7 = 1	Indicator 2 = 1
Data = n, including subfield delimiters	Data = n, including subfield delimiters
	Field Terminator = 1

The effect of this is to require the addition of a count of one to the field length computations output by INFOCAL, to take care of the added field terminators. This should be done in coordination with or subsequent to the actual insertion of the field terminators into the INFOCAL output record.

The starting character position is also a 16-bit binary number. The first directory entry output by INFOCAL will be Tag 000 (= Tag 008) i.e., the fixed length data elements field, and will have a s.c.p. of binary zero which should be converted to 00000. Each successive entry will then be incremented according to the formula

$$scp_{r+1} = scp_{r+1} + r$$

where

scp = Starting Character Position value in the directory entry

r = the ordinal position of the directory entry, i.e., the 1st D/E, the 2nd D/E, ..., _r_th D/E

The parameter in the subscript is "r+1" in the equation, because the s.c.p. of the very first variable field in the record will always be "00000". Each subsequent field is incremented by one position.

If a new field is inserted in a position between the directory and Field 008, the fixed length data elements field, then a different value will have to be used to recompute the s.c.p.'s of the second and all succeeding variable fields. If the newly-inserted field has a length of _i_, then the equation will be

$$scp_{r+1} = scp_{r+1} + (r+i)$$

10.4.4 Modifications to Fixed Length Data Elements Field

10.4.4.1 Structure and Content of MARC II Tag 008 Field

In MARC, the fixed length data elements field is included in a section of the record generically designated "control fields."

Outline of a Control Field

Data Element 1	Data Element 2	Data Element 3	F/T

F/T = Field Terminator

The control fields (Tags 001 - 009) do not use indicators and subfield codes. Data elements in these fields begin in a fixed location relative to the first character position in the field. All control fields end with a field terminator code ($1E_{16}$).

Control Field Tags

001	Control number (Will not be used for Santa Cruz machine file)
008	Fixed length data elements

Content of Tag 008 Field

FIXED LENGTH DATA ELEMENTS

	Name of Data Element	Number of Characters	Character Positions in Field
1.	Date Entered on File	6	0-5
2.	Type of Publication Date Code	1	6
3.	Date 1	4	7-10
4.	Date 2	4	11-14
5.	Country of Publication Code	3	15-17
6.	Illustration Codes	4	18-21
7.	Intellectual Level Code	1	22
8.	Form of Reproduction Code	1	23
9.	Form of Content Codes	4	24-27
10.	Government Publication Indicator	1	28
11.	Conference or Meeting Indicator	1	29
12.	Festschrift Indicator	1	30
13.	Index Indicator	1	31
14.	Main Entry in Body of Entry Indicator	1	32
15.	Fiction Indicator	1	33
16.	Biography Code	1	34
17.	Language Code	3	35-37
18.	Modified Record Indicator	1	38
19.	Cataloging Source Code	1	39

10.4.4.2 Changes in INFOCAL Fixed Length Data Element Field-Translation Table

FIG. 89:
FORMAT TRANSLATION TABLE FIXED LENGTH DATA ELEMENTS SEGMENT

INFOCAL TAG = 000
MARC II TAG = 008

ELEMENT NAME	OLD POS.	ACTION	NEW POS.	CONTENTS, REMARKS, ETC.
1. Date Entered on File	--	MOVE FROM LEADER	0-5	"mmddyy". Formerly in LEADER c.p. 50-55
2. Type of Publ. Date Code	0	CHANGE POSITION	6	Transfer contents
3. Date 1	1-4	CHANGE POSITION	7-10	Transfer contents
4. Date 2	5-8	CHANGE POSITION	11-14	Transfer contents if present
5. Country of Publ.	19-21	CHANGE POSITION TO HERE	15-17	Set to blanks (DEFERRED STATUS)
6. Illus. Codes	22-25	CHANGE POSITION TO HERE	18-21	Transfer contents
7. Intell. Level	--	ADD	22	Set to blank (DEFERRED)
8. Form of Reprod.	9	CHANGE POSITION	23	Set to blank (DEFERRED)
9. Form of Content	10-13	CHANGE POSITION	24-27	Transfer contents
10. Govt. Publ. Indicator	14	CHANGE POSITION	28	Set to blank (DEFERRED)
11. Conference	15	CHANGE POSITION	29	Set to blank (DEFERRED)
12. Festschrift	--	ADD	30	Set to blank (DEFERRED)
13. Index	--	ADD	31	Set to blank (DEFERRED)
14. Main Entry in Body	16	CHANGE POSITION	32	Transfer contents
15. Lit. Group	17	DELETE	--	
16. Cancel Title Added Entry	18	DELETE	--	

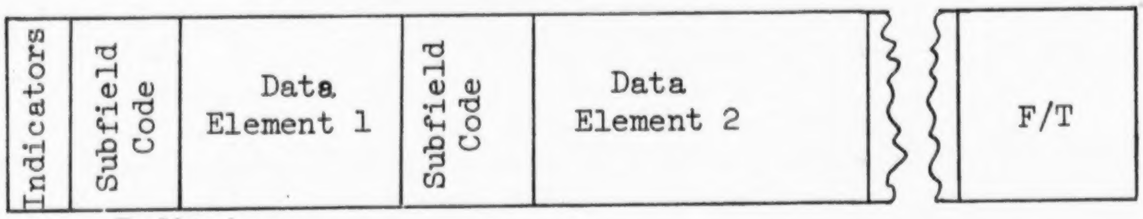
FIG. 89 (Cont.)
FORMAT TRANSLATION TABLE FIXED LENGTH DATA ELEMENTS SEGMENT

ELEMENT NAME	OLD POS.	ACTION	NEW POS.	CONTENTS, REMARKS, ETC.
17. Fiction	--	ADD	33	Set to blank (DEFERRED)
18. Biography	--	ADD	34	Set to blank (DEFERRED)
19. Language Code	--	EXTRACT FROM VARIABLE FIELD TAG 003	35-37	The first and/or only 3-character code is placed here.
20. Modified Record	--	ADD	38	Set to blank (DEFERRED)
21. Cataloging Source	--	ADD	39	b̸ = LC (default)

10.4.5 Modifications to Variable Fields

1. Structure of MARC II Variable Field

Outline of Variable Fields



a. Indicators

Each variable field will begin with two characters which provide descriptive information about the field. The contents of the indicators are specified for the fields in which they are used (see Fig. 88). If the indicators are not used with a particular field, they will contain blanks.

b. Subfield Codes

Variable fields are made up of a single data element or a group of data elements. The subfield code precedes each data element in a field and identifies the data element. The subfield code consists of two characters: a delimiter ($1F_{16}$) followed by a lower case alphabetic character. Subfield codes are specified for each variable field in Section 10.5.4.

c. Data Elements

All data elements in the variable fields may have variable lengths.

d. Field Terminator

The field terminator code is $1E_{16}$.

The list in Fig. 90 contains the variable field tags used in Santa Cruz file output by INFOCAL. These tags will be sequenced in the record directory by the first digit of the tag. Within a block of 100 numbers, however, (e.g., the 600's, subject tracings) the tags will appear in the order in which they appeared in the Santa Cruz file when it was in its original format. The order of fields has not been changed by processing through the INFOCAL program.

2. Reorganizaton of Tagged Fields

a. Reorganization of Language Codes. The language code(s) are stored in a variable field, Tag 003, in the INFOCAL record. This must be altered to the revised MARC II specifications, as follows:

(1) There will always be at least one 3-character language code in each MARC II record. It will be stored in Tag 008, fixed length data elements field, c.p. 35-37. If there is only one code, there will be no variable field.

(2) If the work involves multiple languages or is a translation, there will also be a variable field, Tag 041, to hold the entire language code string. Since the Santa Cruz record in original format may have up to five languages coded, it is possible that this feature will have to be handled in the file translation process. The following action will be taken:

Source Condition (INFOCAL record)	Action
Test for presence of Tag 003 in directory. If present, test length of contents of variable field:	
(a) EQ 3 characters	Move the 3-character code to Tag 008 fixed length data elements field, c.p. 35-37. Purge the directory entry for Tag 003. Purge the variable field.
(b) GT 3 characters	Retain the variable field codes as they appear. Copy the first 3-character code into Tag 008 (but leave it in its original place in the variable field also). Change tag number from 003 to 041 in directory entry. Restructure directory entry as specified in Section 10.4.3. Insert delimiter "$1F_{16}a$" in c.p. 3-4 at head of variable field. Insert "0" in Indicator 1 position in place of INFOCAL value, if necessary, as specified in Fig. 88.
(c) NO Tag 003 present	ERROR Routine: Insert "eng" (lower case) in Tag 008, c.p. 35-37. This is a default value.

b. Reorganization of Imprint Data Elements. The subfield code identifies the constituent data elements of a variable field. For example, the imprint field, Tag 260, may have the following three data elements with their respective subfield codes:

Place - "$1F_{16}$a"
Publisher - "$1F_{16}$b"
Date - "$1F_{16}$c"

These data elements have been coded as tagged fields in the INFOCAL version of MARC II. The coding must now be changed to reflect the latest MARC II specification.

The changes are:

Tag	Disposition
260 Place	No change to tag number. Directory entry must be changed to MARC II format and the value for field length of the reorganized imprint field recomputed to reflect the addition of publisher and place elements.
261 Publisher	Tag number is obsolete and is deleted. The field is dropped as a directory entry and its length is added to that for place in the recomputation of imprint directory entry values. Insert delimiter "$1F_{16}$b" at head of data element. Concatenate with place. Delete extra two spaces for indicators at c.p. 1 and 2 of the obsolete field.
262 Date of Publication	Tag number is obsolete and is deleted. Same treatment as the old 261, except delimiter "$1F_{16}$c" is inserted at head of data element.

Note that as part of the general structural reorganization of the file, a delimiter of "$1F_{16}$a" must be inserted at the head of the variable field for imprint. In this case, it identifies "place."

This will be accomplished by the routines to update INFOCAL if they have not been inserted by the TRANSCOF algorithms done by John Reinke.

c. Reorganization of Subject Subdivisions. The subject subdivisions are stored in variable fields, tagged as follows, in the INFOCAL record:

Tag	Type of Subject Subdivision
655	General subject subdivisions
656	Period subject subdivisions
657	Place subject subdivisions

Only the Tag 655 ("General") has been used in the TRANSCOF program. However, it must be altered as follows, to conform to the revised MARC II specification for the LC subject heading fields:

Tag	Disposition
655	Delete directory entry. Insert "$1F_{16}x$" at head of subject subdivision string. Purge excess spaces at head of field. Concatenate with preceding subject heading field, which will be either a Tag 600 or 650.
600/650	If affected by above subject subdivision, recompute field length of subject heading now including its delimited subdivision.

All of the above special cases will affect the general reorganization action on the directory, and the recomputation of logical record length, etc., specified in part "c" of Section 10.4.3.2.

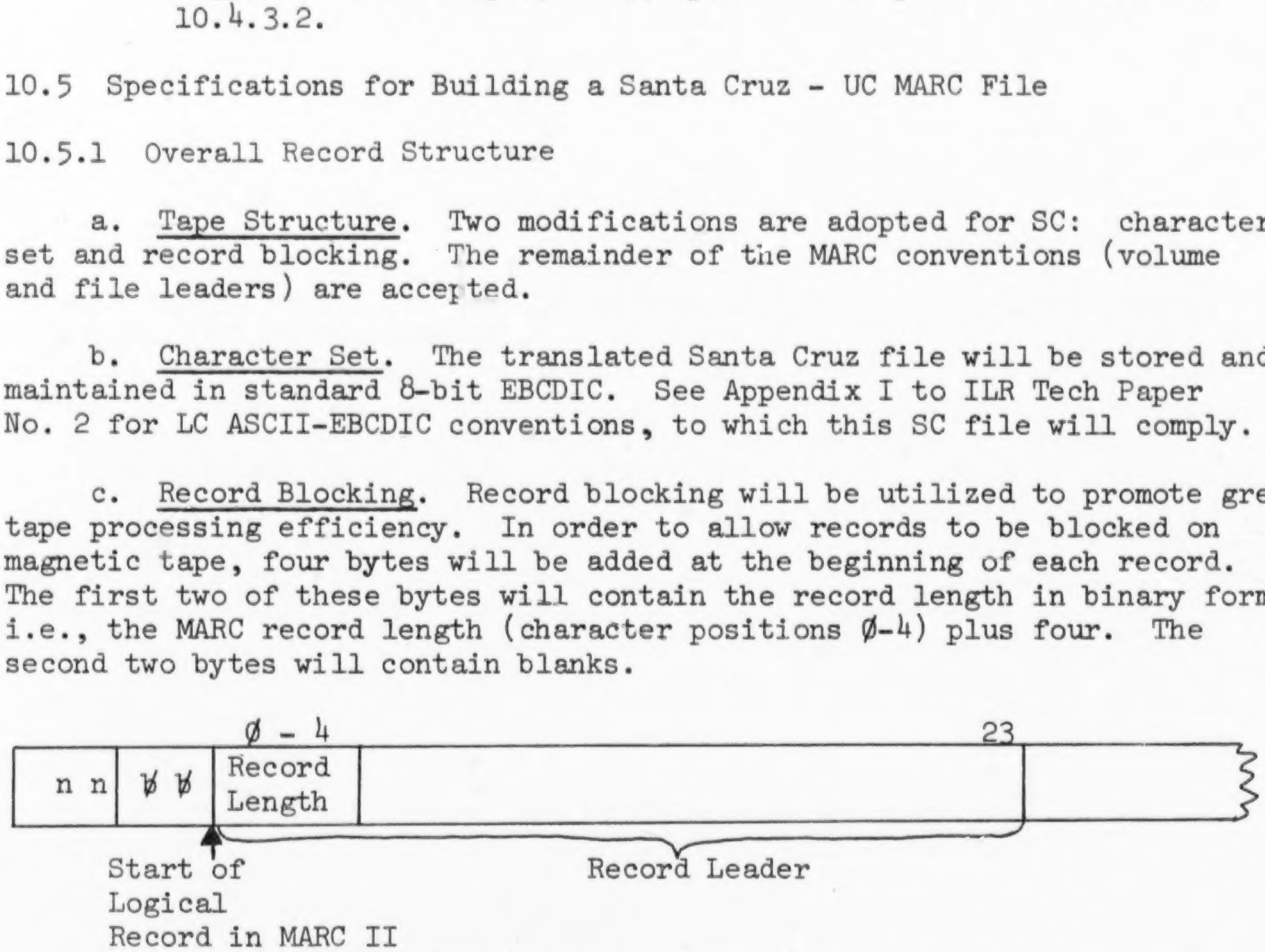
10.5 Specifications for Building a Santa Cruz - UC MARC File

10.5.1 Overall Record Structure

a. Tape Structure. Two modifications are adopted for SC: character set and record blocking. The remainder of the MARC conventions (volume and file leaders) are accepted.

b. Character Set. The translated Santa Cruz file will be stored and maintained in standard 8-bit EBCDIC. See Appendix I to ILR Tech Paper No. 2 for LC ASCII-EBCDIC conventions, to which this SC file will comply.

c. Record Blocking. Record blocking will be utilized to promote greater tape processing efficiency. In order to allow records to be blocked on magnetic tape, four bytes will be added at the beginning of each record. The first two of these bytes will contain the record length in binary form; i.e., the MARC record length (character positions 0-4) plus four. The second two bytes will contain blanks.



nn = total record length in binary form

FIG. 90:
VARIABLE FIELD TAG TRANSLATION TABLE

FIELD NAME	INFOCAL TAG	MARC II TAG
1. Fixed Length Data Elements	000	008
2. Local System Number (INFOCAL Tag "*U" = Tag 091, was used in TRANSCOF program because 035 was not implemented. This is to contain the Shelf Key Data from Santa Cruz CT 000)	091	035
3. Languages (a variable field used only when the work is multi-language or a translation)	003	041
4. Local Call Number (Santa Cruz)	090	NC*
5. Main Entry	1XX	NC
6. Uniform Title (Supplied)	200	240
7. Romanized or Transliterated Title	210	241
8. Title Statement	240	245
9. Edition	250	NC
10. Imprint - Place	260	NC
11. Imprint - Publisher	261	Delimiter: $1F_{16}$b
12. Imprint - Date	262	Delimiter: $1F_{16}$c
13. Collation	300	NC
14. Series Notes	4XX	NC
15. General Note	550	500
16. Bibliography Note	500	504

FIG. 90 (Cont.):
VARIABLE FIELD TAG TRANSLATION TABLE

FIELD NAME	INFOCAL TAG	MARC II TAG
17. Contents Note	520	505
18. Subject Added Entries	6XX	NC
19. Other Added Entries	7XX	NC
20. Series Added Entries	8XX	NC

The IBM Operating System will handle blocking and deblocking functions automatically, with the following conventions:

1. Each block carries four initial characters in the form nn b̸b̸, expressing block size.
2. Logical records are not split between physical blocks.
3. Maximum block size is 3600 characters. Records larger than 3600 characters are unacceptable.
4. Blocks are variable length.

10.5.2 Leader

Record and campus ID codes will be added. Each addition is discussed separately below.

Status (position 5). To comply with UCUCS requirements, insert an EBCDIC "1" to indicate first supplement.

Campus Code (position 8). Used to identify the single source of a record. The following code will be used for the SC translation:

c - Santa Cruz Main University Library

For Local Use (position 9-11). No change from that specified in Fig. 85. The standard MARC II definition will apply to these positions.

Record ID Number (position 17-23). This is the accession number of the record, and it will be used as a basic bibliographic record ID number. The number is carried in EBCDIC representation and is right justified with leading zeros.

No change from that specified in Fig. 85. Number is machine-assigned by ILR program to prepare file for input to TRANSCOF translation program.

10.5.3 Fixed Length Data Elements Field - Tag 007 (UC MARC)

A new control Field, 007, will be added. In the MARC record, a 40-character field (008) contains fixed record descriptors. To avoid redefinition of 008, a new 25-character control field will be defined. It will consist of two areas: common and local.

The common area is defined as follows:

Date (position 0-3). This is defined the same as Date 1 in Field 008; see Subscriber's Guide, p. 33-34.

Language (position 4-6). This is a 3-character code, identica~~l~~ to that used in position 35-37 of Field 008.

Unkeyable Data Indicator (position 7). 0 indicates no unkeyable data in record; 1 indicates presence of unkeyable data in input record. Set "0" in SC file.

"By" Indicator (position 8). 0,1 used to indicate presence/absence of "by" statement in body of card; see position 32 in Field 008.

The local area consists of positions 9-24 and is available for local campus use. Not used in SC file produced from this specification. Fill with blanks. Note that the Santa Cruz file will contain both Fields 007 and 008 in the UC MARC format.

Field 007 will be constructed from data available in Field 008 except where specified above.

10.5.4 Variable Fields - Redefinition of Tag 090 (Local Holdings)

090 Local Holdings. Two basic items of information are to be stored in this field: location and call number. The field consists of Indicator 1 and three subfields.

Indicator 1 - Used to carry campus code, as in position 8 of leader. Insert "c" in Indicator 1 position, for Santa Cruz.

Indicator 2 - Insert a blank for Santa Cruz file.

Subfield Delimiters:

"1F$_{16}$a" - Consists of fixed and variable portion. Fixed portion is 4-characters of the form XX00, where XX is branch code and 00 is number of copies at the branch. Example: "bi04" = biology library, 4 copies. The remainder of the $a subfield is variable and consists of the class portion of call number.

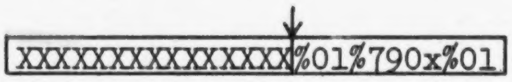
In each Santa Cruz record, the fixed portion will be set to "ƀƀ01". Later, if codes are established for branches at Santa Cruz, it may be possible to extract codes from the original format information stored in c.p. 43-45 of the shelf key card (now Field 091 in the translated file). This code could then be translated into the proper code defined for storage in the UC MARC 090 field, fixed portion.

It is assumed that there will be only one copy of the book, in each case. Again, this may be updated by subsequent processing from the Santa Cruz shelflist information, manual source if not available in the machine file. The call number as found in the INFOCAL record, Tag 090 will then be concatenated to the above fixed portion. All data and coding appended to the INFOCAL field will then be purged. This data portion to be purged begins with a "%" in the INFOCAL record (set by default by INFOCAL).

"$1F_{16}b$" - Book Number. This element will not be identified at this time. The delimiter will not be present.

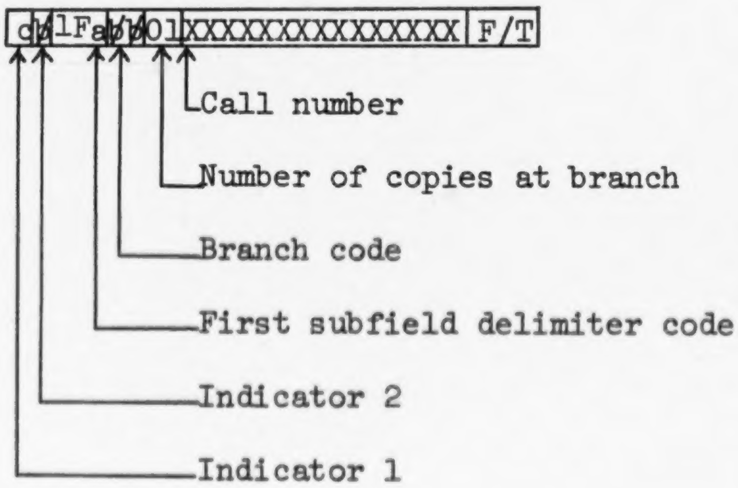
"$1F_{16}c$" - Other Information. This element will not be identified at this time. The delimiter will not be present. (The data will be present as part of the call number for both these last two elements.)

The format of the 090 holdings field as set by default in the INFOCAL program is:


XXXXXXXXXXXXXXX%01%790x%01

All data and coding to be purged are to the right of the arrow, above. The leftmost "%" is also to be purged.

The revised field will look as follows:


c
b̸
1Fa
b̸b̸
01
XXXXXXXXXXXXXXXX
F/T
Call number
Number of copies at branch
Branch code
First subfield delimiter code
Indicator 2
Indicator 1